



PDAL: Point cloud Data Abstraction Library

Release 1.6.0

**Howard Butler
Brad Chambers
Michael Gerlek
PDAL Contributors**

Nov 16, 2017

CONTENTS

1 News	3
1.1 10-12-2017	3
2 About	5
2.1 About	5
3 Download	13
3.1 Download	13
4 Quickstart	15
4.1 Quickstart	15
5 Applications	23
5.1 Applications	23
6 Community	39
6.1 Community	39
7 Drivers	41
7.1 Pipeline	41
7.2 Readers	50
7.3 Writers	79
7.4 Filters	104
8 Dimensions	185
8.1 Dimensions	185
9 Python	189
9.1 Python	189
10 Tutorials	193
10.1 Tutorials	193
11 Workshop	239

11.1 Point Cloud Processing and Analysis with PDAL	239
12 Development	323
12.1 Development	323
12.2 API	386
12.3 FAQ	451
12.4 License	452
12.5 References	453
13 Indices and tables	455
Bibliography	457
Index	459



PDAL is a C++ [BSD](http://www.opensource.org/licenses/bsd-license.php) (<http://www.opensource.org/licenses/bsd-license.php>) library for translating and manipulating [point cloud data](http://en.wikipedia.org/wiki/Point_cloud) (http://en.wikipedia.org/wiki/Point_cloud). It is very much like the [GDAL](http://www.gdal.org) (<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 50) and [Writers](#) (page 79) to list data formats it supports, and see [Filters](#) (page 104) 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 23) provides more information on that topic.

Finally, PDAL speaks Python by both embedding and extending it. Visit [Python](#) (page 189) 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>

**CHAPTER
ONE**

NEWS

1.1 10-12-2017

PDAL 1.6.0 has been released. Visit [Download](#) (page 13) to obtain a copy of the source code, or follow the [Quickstart](#) (page 15) to get going in a hurry with [Docker](#) (<https://www.docker.com/>).

CHAPTER TWO

ABOUT

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 104) on point clouds into *pipelines* (page 41) 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 59) stage, a [Reprojection](#) (page 173) stage, and a [PostgreSQL Writer](#) (page 98), 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.



Fig. 2.1: A simple PDAL pipeline composed of a reader, filter, and writer stages.

A PDAL JSON [Pipeline](#) (page 41) that composes this operation to reproject and load the data into PostgreSQL might look something like the following:

```
1 {
2     "pipeline": [
3         {
4             "type": "readers.las",
5             "filename": "input.las"
6         },
7         {
8             "type": "filters.reprojection",
9             "out_srs": "EPSG:3857"
10        },
11        {
12            "type": "writers.pgpointcloud",
13            "connection": "host='localhost' dbname='lidar' user='hobu'",
14            "table": "output",
15            "srid": "3857"
16        }
17    ]
18 }
```

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 31), 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](#)

(http://www.gdal.org/gdal_vrttut.html).

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) (<http://lastools.org>) from [Martin Isenburg](https://www.cs.unc.edu/~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) (<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) (<http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html>). [LAStools](http://lastools.org) (<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 185) types provided by the LAS format.
4. PDAL is coordinated by users with its declarative [JSON](#) (page 41) 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) (<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 links and uses PCL, and PDAL provides a convenient pipeline mechanism to orchestrate PCL operations.

Note: See [Filtering data with PCL](#) (page 209) for more detail on how to take advantage of PCL capabilities within PDAL operations.

Greyhound and Entwine

Greyhound (<http://greyhound.io>) is an open source software from Hobu, Inc. (<https://hobu.co>) that allows clients to query and stream progressive point cloud data over the network. Entwine (<https://entwine.io>) is open source software from Hobu, Inc. that organizes massive point cloud collections into Greyhound (<http://greyhound.io>)-streamable data services. These two software projects allow province-scale LiDAR collections to be organized and served via HTTP clients over the internet. PDAL provides *readers.greyhound* (page 56) to allow users to read data into PDAL processes from that server.

plas.io and Potree

plas.io (<http://plas.io>) 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 for it at plasiojs.io and <https://github.com/hobu/plasio-ui>

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: Both renderers can now consume data from Greyhound. See them in action at <http://speck.ly> and <http://potree.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 pre-dates 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) (<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/) (<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-Sheet-Article-View/Article/476649/remote-sensinggeographic-information-systems-center/) (<http://www.erdc.usace.army.mil/Media/Fact-Sheets/Fact-Sheet-Article-View/Article/476649/remote-sensinggeographic-information-systems-center/>) [GRiD](http://lidar.io/) (<http://lidar.io/>) project. Functionality that was creeping into [libLAS](http://liblas.org/) (<http://liblas.org/>) was pulled into a new library, and it was designed from the ground up to mimic successful extract, transform, and load libraries in the geospatial software domain. PDAL has steadily attracted more contributors as other software developers use it to provide point cloud data translation and processing capability to their software.

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 185).

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 104), [Readers](#) (page 50), and [Writers](#) (page 79). 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 39) 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 23) 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 386) 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 and its [source code](#) (<https://github.com/PDAL/PDAL/tree/master/test/unit>).

See also:

PDAL *software* (page 193) *development* (page 355) *tutorials* (page 368) 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 23) 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 27) about a data set
- Data *translation* (page 36) 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 26)
- *Merge* (page 30) or *split* (page 33) data
- *Catalog* (page 34) collections of data

Note: The command line utilities are often simply *pipeline* (page 31) and *Pipeline* (page 41) 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 189) document contains information on how to install and use the PDAL Python extension.

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.

DOWNLOAD

3.1 Download

Contents

- *Download* (page 13)
 - *Current Release(s)* (page 13)
 - *Past Releases* (page 14)
 - *Development Source* (page 14)
 - *Binaries* (page 14)
 - * *Docker* (page 14)
 - * *Windows* (page 14)
 - * *RPMs* (page 14)
 - * *Debian* (page 14)

3.1.1 Current Release(s)

- **2017-10-12 PDAL-1.6.0-src.tar.gz**
(<http://download.osgeo.org/pdal/PDAL-1.6.0-src.tar.gz>) Release Notes
(<https://github.com/PDAL/PDAL/releases/tag/1.6.0>) ([md5](#)
(<http://download.osgeo.org/pdal/PDAL-1.6.0-src.tar.gz.md5>))

3.1.2 Past Releases

- **2017-04-06** PDAL-1.5.0-src.tar.gz
(<http://download.osgeo.org/pdal/PDAL-1.5.0-src.tar.gz>)
- **2016-12-15** PDAL-1.4.0-src.tar.gz
(<http://download.osgeo.org/pdal/PDAL-1.4.0-src.tar.gz>)

3.1.3 Development Source

The main repository for PDAL is located on github at <https://github.com/OSGeo/OSGeo4W>

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

```
git clone https://github.com/OSGeo/OSGeo4W.git pdal
```

3.1.4 Binaries

Docker

The fastest way to get going with PDAL is to use the Docker build. See the *Docker tutorial* (page 15) for more information.

```
docker pull pdal/pdal:1.6
```

Windows

Windows builds are available via OSGeo4W (64-bit only). See the [workshop-osgeo4w](#) page 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>).

CHAPTER FOUR

QUICKSTART

4.1 Quickstart

4.1.1 Introduction

It's a giant pain to build everything yourself. The quickest way to start using PDAL is to leverage builds that were constructed by the PDAL development team using *Docker* (page 15). Docker is a containerization technology that allows you to run pre-built software in a way that is isolated from your system. Think of it like a binary that doesn't depend on your operating system's configuration to be able to run.

This exercise will print the first point of an *ASPRS LAS* (page 59) file. It will utilize the PDAL *command line application* (page 23) to inspect the file.

Note: While Docker is convenient, it is not for everyone. You can also obtain the software by installing a Linux package from *Download* (page 13) or compiling it yourself from *Compilation* (page 334).

Docker is not required to use PDAL, and there are packages available on Linux (Debian, RPM) and OSX ([Homebrew](http://brew.sh) (<http://brew.sh>))). See *Download* (page 13) to obtain them. If you are a developer looking to leverage PDAL, you will need access to the library in your environment, but this quick start document is for those looking to quickly interact with data using PDAL's *command line applications* (page 23) and *Pipeline* (page 41).

If you need to compile your own copy of PDAL, see *Compilation* (page 334) for more details.

4.1.2 Install Docker

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

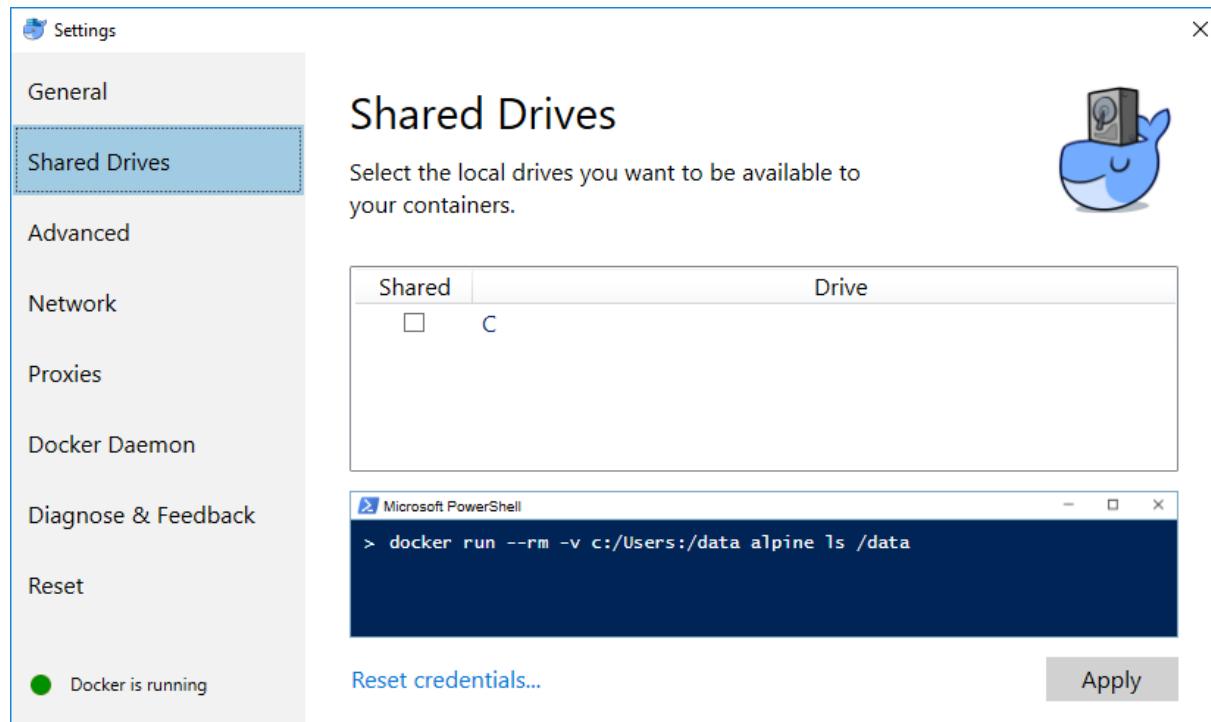
- [Windows](https://docs.docker.com/docker-for-windows/) (<https://docs.docker.com/docker-for-windows/>)

- OSX (<https://docs.docker.com/docker-for-mac/>)
- Linux (<https://docs.docker.com/engine/installation/linux/>)

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

Enable Docker access to your machine

In order for Docker to be able to interact with data on your machine, you must make sure to tell it to be able to read your drive(s). Right-click on the little Docker whale icon in your System Tray, choose Settings, and click the Shared box by your C drive:



Run Docker Quickstart Terminal

Docker (page 15) is most easily accessed using a terminal window that it configures with environment variables and such. Run PowerShell or *cmd.exe* and issue a simple *info* command to verify that things are operating correctly:

```
docker info
```



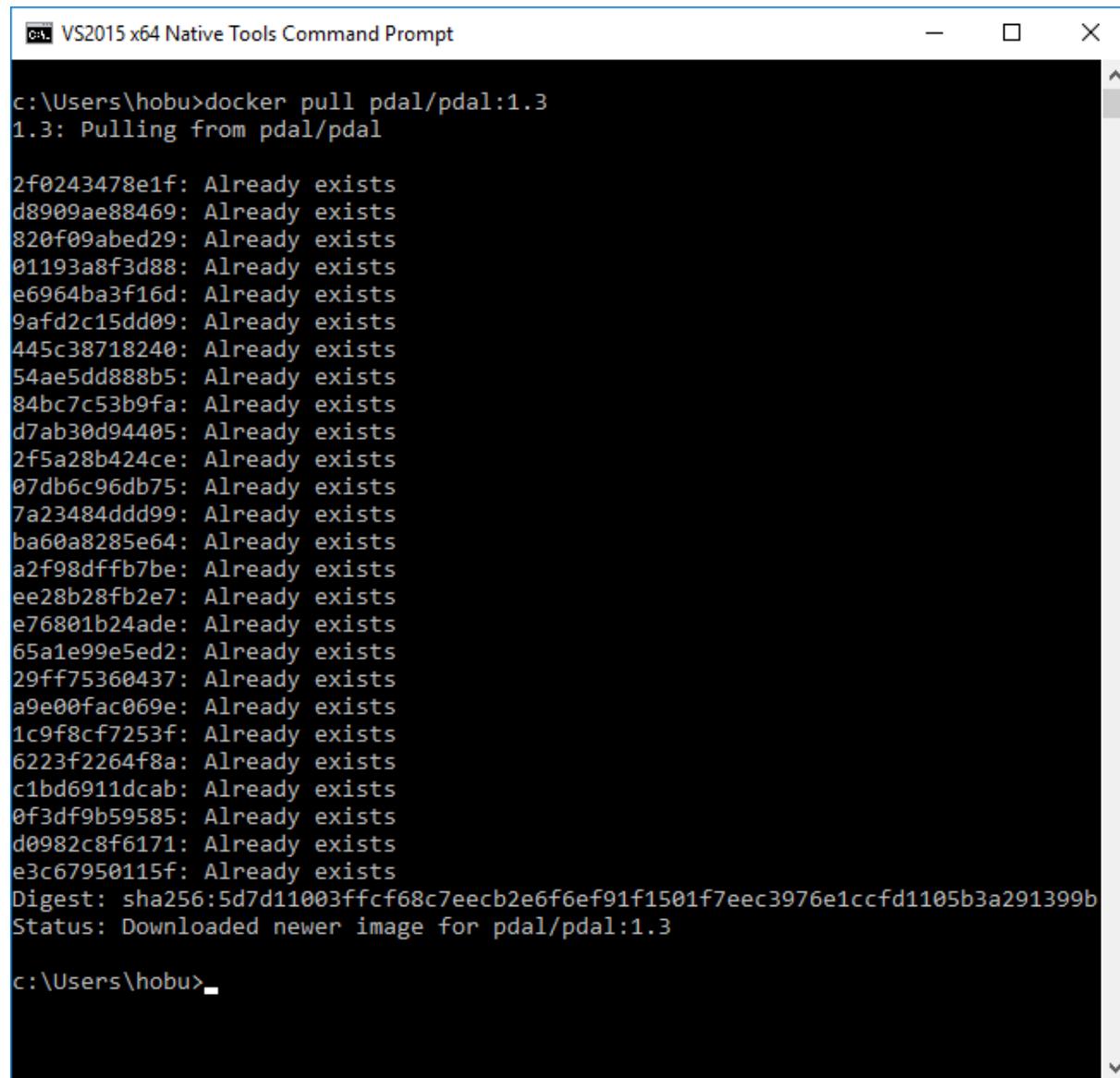
```
VS2015 x64 Native Tools Command Prompt
C:\>docker info
Containers: 0
Running: 0
Paused: 0
Stopped: 0
Images: 0
Server Version: 1.12.0
Storage Driver: aufs
Root Dir: /var/lib/docker/aufs
Backing Filesystem: extfs
Dirs: 0
Dirperm1 Supported: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
Volume: local
Network: host bridge null overlay
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Security Options: seccomp
Kernel Version: 4.4.15-moby
Operating System: Alpine Linux v3.4
OSType: linux
Architecture: x86_64
CPUs: 2
Total Memory: 986.8 MiB
Name: moby
ID: DHFS:DF5Q:4HD3:AFTC:BAI5:XPAV:EEZ6:JFGI:6Y4J:EEVH:EHHV:6YKU
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
Insecure Registries:
 127.0.0.0/8

C:\>
```

Obtain PDAL Image

A PDAL image based on the latest release, including all recent patches, is pushed to [Docker Hub](#) (<http://hub.docker.com>) with every code change on the PDAL maintenance branch (find out more about that at [here](#)). We need to pull it locally so we can use it to run PDAL commands. Once it is pulled, we don't have to pull it again unless we want to refresh it for whatever reason.

```
docker pull pdal/pdal:1.6
```



The screenshot shows a terminal window titled "VS2015 x64 Native Tools Command Prompt". The command "docker pull pdal/pdal:1.6" was run, and the output is displayed. The output shows that the image already exists for many layers, and a newer version was downloaded. The final status message is "Status: Downloaded newer image for pdal/pdal:1.3".

```
c:\Users\hobu>docker pull pdal/pdal:1.6
1.6: Pulling from pdal/pdal
2f0243478e1f: Already exists
d8909ae88469: Already exists
820f09abed29: Already exists
01193a8f3d88: Already exists
e6964ba3f16d: Already exists
9afcd2c15dd09: Already exists
445c38718240: Already exists
54ae5dd888b5: Already exists
84bc7c53b9fa: Already exists
d7ab30d94405: Already exists
2f5a28b424ce: Already exists
07db6c96db75: Already exists
7a23484ddd99: Already exists
ba60a8285e64: Already exists
a2f98dfffb7be: Already exists
ee28b28fb2e7: Already exists
e76801b24ade: Already exists
65a1e99e5ed2: Already exists
29ff75360437: Already exists
a9e00fac069e: Already exists
1c9f8cf7253f: Already exists
6223f2264f8a: Already exists
c1bd6911dcab: Already exists
0f3df9b59585: Already exists
d0982c8f6171: Already exists
e3c67950115f: Already exists
Digest: sha256:5d7d11003ffcf68c7eecb2e6f6ef91f1501f7eec3976e1ccfd1105b3a291399b
Status: Downloaded newer image for pdal/pdal:1.3

c:\Users\hobu>
```

Note: Other PDAL versions are provided at the same [Docker Hub](http://hub.docker.com) (<http://hub.docker.com>) location, with an expected tag name (ie `pdal/pdal:1.6`, or `pdal/pdal:1.x`) for major PDAL versions. The PDAL Docker hub location at <https://hub.docker.com/u/pdal/> has images and more information on this topic.

4.1.3 Fetch Sample Data

We need some sample data to play with, so we're going to download the `autzen.laz` file to your `C:/Users/hobu/Downloads` fold. Inside your terminal, issue the following command:

```
explorer.exe http://www.libblas.org/samples/autzen/autzen.laz
```

```
cd C:/Users/hobu/Downloads  
copy autzen.laz ..
```

4.1.4 Print the first point

```
docker run -v /c/Users/hobu:/data pdal/pdal:1.6 pdal info /data/  
autzen.laz -p 0
```

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

1. docker: We are running PDAL within the context of docker, so all of our commands will start with the `docker` command.
2. run: Tells docker we're going to run an image
3. `-v /c/Users/hobu:/data`: Maps our home directory to a directory called `/data` inside the container.

See also:

The [Docker Volume](https://docs.docker.com/engine/userguide/dockervolumes/) (<https://docs.docker.com/engine/userguide/dockervolumes/>) document describes mounting volumes in more detail.

4. `pdal/pdal:1.6`: This is the Docker image we are going to run. We fetched it with the command above. If it were not already fetched, Docker would attempt to fetch it when we run this command.
5. `pdal`: We're finally going to run the `pdal` command :)
6. `info`: We want to run `info` (page 27) on the data
7. `/data/autzen.laz`: The `pdal` command is now running in the context of our container, which we mounted a `/data` directory in with the volume mount operation in Step #3. Our `autzen.laz` file resides there.



```
c:\Users\hobu>docker run -v /c/Users/hobu:/data pdal/pdal:1.3 pdal info /data/autzen.laz -p 0
{
  "filename": "\data\autzen.laz",
  "pdal_version": "1.3.0 (git-version: 3075df)",
  "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
    }
  }
}

c:\Users\hobu>
```

4.1.5 What's next?

- Visit [Applications](#) (page 23) to find out how to utilize PDAL applications to process data on the command line yourself.
- Visit [Development](#) (page 323) to learn how to embed and use PDAL in your own applications.
- [Readers](#) (page 50) lists the formats that PDAL can read, [Filters](#) (page 104) lists the kinds of operations you can do with PDAL, and [Writers](#) (page 79) lists the formats PDAL can write.
- [Tutorials](#) (page 193) contains a number of walk-through tutorials for achieving many tasks with PDAL.

- *The PDAL workshop* (page 239) contains numerous hands-on examples with screenshots and example data of how to use PDAL *Applications* (page 23) to tackle point cloud data processing tasks.
- *Python* (page 189) describes how PDAL embeds and extends Python and how you can leverage these capabilities in your own programs.

See also:

Community (page 39) is a good source to reach out to when you're stuck.

CHAPTER FIVE

APPLICATIONS

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 < myxml.xml
```

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
```

Note: Driver-specific options can be identified using the pdal <command> --help invocation.

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'  
-----  
→-----  
-----  
Dimension      X           Y           Z  
-----  
Min            0.0000       0.0000       0.0000  
Max            0.0000       0.0000       0.0000  
Mean           0.0000       0.0000       0.0000  
-----
```

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 134).

Note: The density command is only available when PDAL is linked with Hexer (BUILD_PLUGIN_HEXBIN=ON).

```
$ 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 diff

The diff command is used for executing a simple contextual difference between two sources.

```
$ pdal diff <source> <candidate>
```

```
--source               source file name
--candidate            candidate file name
--output                output file name
--2d                   only 2D comparisons/indexing
--detail                Output deltas per-point
--alldims              Compute diffs for all dimensions (not just X, Y, Z)
```

The command returns 0 and produces no output if the files describe the same point data in the same format, otherwise 1 is returned and a JSON-formatted description of the differences is produced.

The command checks for the equivalence of the following items:

- Different schema
- Expected count
- Metadata
- Actual point count
- Byte-by-byte point data

5.1.4 ground

The `ground` command is used to segment the input point cloud into ground versus non-ground returns.

```
$ 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
--initial_distance   Initial distance
--cell_size           Cell size
--classify            Apply classification labels?
--extract             extract ground returns?
--approximate, -a    use approximate algorithm? (much faster)
```

```
For more information, see the full documentation for PDAL at http://pdal.io/
```

5.1.5 hausdorff

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>
```

```
--source arg      Non-positional option for specifying filename of source file.  
--candidate arg  Non-positional option for specifying filename to test against source.
```

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"
  ],
  "hausdorff": 1.303648726,
  "pdal_version": "1.3.0 (git-version: 191301)"
}
```

Note: The hausdorff is computed for XYZ coordinates only and as such says nothing about differences in other dimensions or metadata.

5.1.6 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, 10, 100-200)
→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 file)
→dataset)	
--boundary	Compute a hexagonal hull/boundary of the dataset
→dataset	
--dimensions	Dimensions on which to compute statistics
--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": 245383.38808001476,
    "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": 425.8899999999999
},
...
}
```

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":
    {
        "point":
        [
            {
                "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": 431.6600000000003  
,  
{  
    "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": 636896.32999999996,  
    "Y": 849087.70000000007,  
    "Z": 446.3899999999999  
,  
    ...  
}
```

5.1.7 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.

5.1.8 pcl

The `pcl` command is used to invoke a PCL JSON pipeline. See *Filtering data with PCL* (page 209) for more information.

Note: The `pcl` command is only available when PDAL is linked with PCL.

```
$ pdal pcl <input> <output> <pcl>
```

-- <code>input</code> , -i	Input filename
-- <code>output</code> , -o	Output filename
-- <code>pcl</code> , -p	PCL filename
-- <code>compress</code> , -z	Compress output data (<code>if</code> supported by output <code>format</code>)
-- <code>metadata</code> , -m	Forward metadata (VLRs, header entries, etc) <code>from previous</code>

5.1.9 pipeline

The `pipeline` command is used to execute *Pipeline* (page 41) JSON. See *Reading with PDAL* (page 193) or *Pipeline* (page 41) for more information.

```
$ pdal pipeline <input>
```

-- <code>input</code> , -i	Input filename
-- <code>pipeline-serialization</code>	Output file <code>for</code> pipeline serialization
-- <code>validate</code>	Validate the pipeline (including <code>serialization</code>), but do <code>not</code> write points
-- <code>progress</code>	Name of file <code>or</code> FIFO to which stages <code>should write</code> progress information. The file/FIFO must exist. PDAL will <code>not</code> <code>create the</code> <code>progress file.</code>
-- <code>stdin</code> , -s	Read pipeline <code>from standard input</code>
-- <code>stream</code>	Attempt to run pipeline <code>in</code> streaming mode.
-- <code>metadata</code>	Metadata filename

Substitutions

The `pipeline` command can accept command-line option substitutions and they replace existing options that are specified in the input JSON pipeline. If multiple stages of the same

name exist in the pipeline, *all* stages would be overridden. 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
```

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"
    },
    "placeholder.laz"
  ]
}

$ pdal pipeline translate.json --writers.las.filename=output.laz \
--stage.las1.filename=file1.las --stage.las2.filename=file2.las
```

Options specified by tag names override options specified by stage types.

5.1.10 random

The `random` command is used to create a random point cloud. It uses [readers.faux](#) (page 51) 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>
```

```
--output, -o          Output file name
--compress, -z        Compress output data (if supported by output)
--format)              How many points should we write?
```

```
--bounds          Extent (in XYZ to clip output to)
--mean           A comma-separated or quoted, space-separated list
  ↵of means
    (normal mode): --mean 0.0,0.0,0.0 --mean "0.0 0.0 0.0"
--stdev          A comma-separated or quoted, space-separated list
  ↵of
    standard deviations (normal mode): --stdev 0.0,0.0,0.0 --stdev
  ↵"0.0 0.0 0.0"
--distribution   Distribution (uniform / normal)
```

5.1.11 sort

The `sort` command uses *filters.mortonorder* (page 148) 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
```

5.1.12 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:

```
$ 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.

5.1.13 tindex

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

--tindex	OGR-readable/writeable tile index output
--filespec	Build: Pattern of files to index. Merge: ↳
↳Output filename	
--fast_boundary	Use extent instead of exact boundary
--lyr_name	OGR layer name to write into datasource
--tindex_name	Tile index column name
--ogrdriver, -f	OGR driver name to use
--t_srs	Target SRS of tile index
--a_srs	Assign SRS of tile with no SRS to this value
--write_absolute_path	Write absolute rather than relative file paths
--merge	Whether we're merging the entries in a tindex: ↳
↳file.	
--stdin, -s	Read filespec pattern from standard input

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

```
$ 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.

```
--tindex      OGR-readable/writeable tile index output
--filespec    Build: Pattern of files to index. Merge: Output ↴
--filename
--lyr_name    OGR layer name to write into datasource
--tindex_name  Tile index column name
--ogrdriver, -f OGR driver name to use
--t_srs        Target SRS of tile index
--bounds       Extent (in XYZ) to clip output to
--polygon      Well-known text of polygon to clip output
```

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`:

```
$ find las/ -iname "*.las" | pdal tindex 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 index.sqlite "*.las" -f "SQLite" --lyr_name "pdal" \
--t_srs "EPSG:4326"
```

5.1.14 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.

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

--input, -i	Input filename
--output, -o	Output filename
--filter, -f	Filter type
--json	PDAL pipeline from which to extract filters.
--pipeline, -p	Pipeline output
--metadata, -m	Dump metadata output to the specified file
--reader, -r	Reader type
--writer, -w	Writer type

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 be 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 lines. If a reader/writer stage referenced 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" \
    --filters.crop.polygon="POLYGON ((636889.412951239268295 851528.
    ↪512293258565478 422.7001953125,636899.14233423944097 851475.
    ↪000686757150106 422.4697265625,636899.14233423944097 851475.
    ↪000686757150106 422.4697265625,636928.33048324030824 851494.
    ↪459452757611871 422.5400390625,636928.33048324030824 851494.
    ↪459452757611871 422.5400390625,636928.33048324030824 851494.
    ↪459452757611871 422.5400390625,636976.977398241520859 851513.
    ↪918218758190051 424.150390625,636976.977398241520859 851513.
    ↪918218758190051 424.150390625,637069.406536744092591 851475.
    ↪000686757150106 438.7099609375,637132.647526245797053 851445.
    ↪812537756282836 425.9501953125,637132.647526245797053 851445.
    ↪812537756282836 425.9501953125,637336.964569251285866 851411.
    ↪759697255445644 425.8203125,637336.964569251285866 851411.
    ↪759697255445644 425.8203125,637473.175931254867464 851158.
    ↪795739248627797 435.6298828125,637589.928527257987298 850711.
    ↪244121236610226 420.509765625,637244.535430748714134 850511.
    ↪791769731207751 420.7998046875,636758.066280735656619 850667.
    ↪461897735483944 434.609375,636539.155163229792379 851056.
    ↪63721774588339 422.6396484375,636889.412951239268295 851528.
    ↪512293258565478 422.7001953125))" \
        ./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 PCL-based voxel grid filter, followed by the PMF filter and a range filter. We can even set stage-specific parameters as shown.

```
$ pdal translate input.las output.las pclblock pmf range \
--filters.pclblock.methods="[{\"name\":\"VoxelGrid\"}]" \
--filters.pmf.approximate=true \
--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”.

```
$ 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”.

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

CHAPTER SIX

COMMUNITY

6.1 Community

PDAL's community interacts through *Mailing List* (page 39), *GitHub* (page 39), *Gitter* (<https://gitter.im/PDAL/PDAL>) and *IRC* (page 40). Please feel welcome to ask questions and participate in all of the venues. The *Mailing List* (page 39) communication channel is for general questions, development discussion, and feedback. The *GitHub* (page 39) communication channel is for development activities, bug reports, and testing. The *IRC* (page 40) 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_\(software\)](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 323) 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) (<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 40). Gitter uses your [GitHub](#) (page 39) 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) (<http://freenode.net>). This mechanism is usually reserved for active meetings and other outreach with the community. The [Mailing List](#) (page 39) and [GitHub](#) (page 39) avenues are going to be more productive communication channels in most situations.

DRIVERS

7.1 Pipeline

Pipelines are the operative construct in PDAL, and it is how data are modeled from reading, processing, and writing. PDAL internally constructs a pipeline to perform data translation operations using *translate* (page 36), for example. While specific *applications* (page 23) are useful in many contexts, a pipeline provides useful advantages for more complex things:

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/) (<http://www.json.org/>) manipulation facilities of whatever language you want.

Note: *pipeline* (page 31) is used to invoke pipeline operations via the command line.

Warning: As of PDAL 1.2, [JSON](http://www.json.org/) (<http://www.json.org/>) is the preferred specification language for PDAL pipelines. XML was dropped at the 1.6 release.

7.1.1 Introduction

A JSON object represents a PDAL processing pipeline. The structure is always a JSON object, with the primary object called `pipeline` being an array of inferred or explicit PDAL *Stage Objects* (page 44) representations.

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:

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



Fig. 7.1: A simple pipeline to convert *LAS* (page 59) to *BPF* (page 50) 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 A1, merges the result with B, and writes the merged output to a GeoTIFF file with the *writers.gdal* (page 81) writer:

```
{
  "pipeline": [
    {
      "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",
  "inputs": [
    "A2",
    "B"
  ]
},
{
  "type": "writers.gdal",
  "filename": "output.tif"
}
]
```



Fig. 7.2: A more complex pipeline that merges two inputs together but uses *filters.reprojection* (page 173) to transform the coordinate system of file B.las from UTM (<http://spatialreference.org/ref/epsg/nad83-utm-zone-16n/>) to Geographic (<http://spatialreference.org/ref/epsg/4326/>).

7.1.2 Pipeline Objects

PDAL JSON pipelines always consist of a single object. This object (referred to as the PDAL JSON object below) represents a processing pipeline.

- The PDAL JSON object may have any number of members (name/value pairs).
- The PDAL JSON object must have a *Pipeline Array* (page 44).

Pipeline Array

- The pipeline array may have any number of string or *Stage Objects* (page 44) 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 50), *Writers* (page 79), and *Filters* (page 104).

- 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.
- Applications can place a `user_data` node on any stage object and it will be carried through to any serialized pipeline output.

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).

7.1.3 Extended Examples

BPF to LAS

The following pipeline converts the input file from *BPF* (page 50) to *LAS* (page 85), inferring both the reader and writer type, and setting a number of options on the writer stage.

```
{
  "pipeline": [
    "utm15.bpf",
    {
      "filename": "out2.las",
      "scale_x": 0.01,
      "offset_x": 311898.23,
      "scale_y": 0.01,
      "offset_y": 4703909.84,
      "scale_z": 0.01,
      "offset_z": 7.385474
    }
  ]
}
```

Python HAG

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 164) 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 (page 129) describes using a specific filter to do this job in more detail.

```
{
  "pipeline": [
    "autzen.las",
    {
      "type": "ferry",
      "dimensions": "Z=HAG"
    },
    {
      "type": "filters.python",
      "script": "hag.py",
      "function": "filter",
    }
  ]
}
```

```
        "module": "anything"
    },
    "autzen-hag.las"
]
}
```

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.pmf](#) (page 159), filters out all points but the ground returns (classification value of 2) using the [filters.range](#) (page 170), and then creates the DTM using the [writers.gdal](#) (page 81).

```
{
  "pipeline": [
    "autzen-full.las",
    {
      "type": "ground",
      "approximate": true,
      "max_window_size": 33,
      "slope": 1.0,
      "max_distance": 2.5,
      "initial_distance": 0.15,
      "cell_size": 1.0
    },
    {
      "type": "range",
      "limits": "Classification[2:2]"
    },
    {
      "type": "writers.gdal",
      "filename": "autzen-surface.tif",
      "output_type": "min",
      "output_format": "tif",
      "grid_dist_x": 1.0,
      "grid_dist_y": 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.

```
{
  "pipeline": [
    {
      "filename": "1.2-with-color.las",
      "spatialreference": "EPSG:2993"
    },
    {
      "type": "decimation",
      "step": 2,
      "offset": 1
    },
    {
      "type": "colorization",
      "raster": "autzen.tif",
      "dimensions": "Red:1:1, Green:2:1, Blue:3:1"
    },
    {
      "filename": "junk.txt",
      "delimiter": ",",
      "write_header": false
    }
  ]
}
```

Merge & Reproject

Our first example with multiple readers, this pipeline infers the reader types, and assigns spatial reference information to each. Next, the [filters.merge](#) (page 145) merges points from all previous readers, and the [filters.reprojection](#) (page 173) filter reprojects data to the specified output spatial reference system.

```
{
  "pipeline": [
    {
      "filename": "1.2-with-color.las",
      "spatialreference": "EPSG:2027"
    },
    {
      "filename": "1.2-with-color.las",
      "spatialreference": "EPSG:2027"
    },
    {
      "type": "filters.merge"
    }
  ]
}
```

```
{  
    "type": "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.

```
{  
    "pipeline": [  
        "/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.

7.1.4 API Considerations

A *Pipeline* is composed as an array of *pdal::Stage* (page 420) , with the first stage at the beginning and the last at the end. There are two primary building blocks in PDAL, *pdal::Stage* (page 420) and *pdal::PointView* (page 413). *pdal::Reader* (page 420), *pdal::Writer* (page 444), and *pdal::Filter* (page 404) are all subclasses of *pdal::Stage* (page 420).

pdal::PointView (page 413) is the substrate that flows between stages in a pipeline and transfers the actual data as it moves through the pipeline. A *pdal::PointView* (page 413) contains a *pdal::PointTablePtr*, which itself contains a list of *pdal::Dimension* (page 395) objects that define the actual channels that are stored in the *pdal::PointView* (page 413).

PDAL provides four types of stages – *pdal::Reader* (page 420), *pdal::Writer* (page 444), *pdal::Filter* (page 404), and *pdal::MultiFilter* – with the latter being hardly used (just *filters.merge* (page 145)) at this point. A Reader is a producer of data, a Writer is a consumer of data, and a Filter is an actor on data.

Note: As a C++ API consumer, you are generally not supposed to worry about the underlying storage of the PointView, but there might be times when you simply just “want the data.” In those situations, you can use the `pdal::PointView::getBytes()` method to stream out the raw storage.

Usage

While pipeline objects are manipulable through C++ objects, the other, more convenient way is through an JSON syntax. The JSON syntax mirrors the arrangement of the Pipeline, with options and auxiliary metadata added on a per-stage basis.

We have two use cases specifically in mind:

- a *command-line* (page 31) application that reads an JSON file to allow a user to easily construct arbitrary writer pipelines, as opposed to having to build applications custom to individual needs with arbitrary options, filters, etc.
- a user can provide JSON for a reader pipeline, construct it via a simple call to the PipelineManager API, and then use the `pdal::Stage::read()` function to perform the read and then do any processing of the points. This style of operation is very appropriate for using PDAL from within environments like Python where the focus is on just getting the points, as opposed to complex pipeline construction.

```
{  
  "pipeline": [  
    "/path/to/my/file/input.las",  
    "output.las"  
  ]  
}
```

Note: <https://github.com/PDAL/PDAL/blob/master/test/data/pipeline/> contains test suite pipeline files that provide an excellent example of the currently possible operations.

Stage Types

`pdal::Reader` (page 420), `pdal::Writer` (page 444), and `pdal::Filter` (page 404) are the C++ classes that define the stage types in PDAL. Readers follow the pattern of `readers.las` (page 59) or `readers.oci` (page 66), Writers follow the pattern of `writers.las` (page 85) or `writers.oci` (page 66), with Filters using `filters.reprojection` (page 173) or `filters.crop` (page 118).

Note: stage_index contains a full listing of possible stages and descriptions of their options.

Note: Issuing the command pdal info --options will list all available stages and their options. See [info](#) (page 27) for more.

7.2 Readers

Readers provide [Dimensions](#) (page 185) to [Pipeline](#) (page 41). 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 76), a complex database like [readers.oci](#) (page 66), or a network service like [readers.greyhound](#) (page 56).

7.2.1 readers.bpf

BPF is an NGA specification for point cloud data. The specification can be found at <https://nsgreg.nga.mil/doc/view?i=4220&month=8&day=30&year=2016>. 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

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

Options

filename BPF file to read [Required]

count Maximum number of points to read [Optional]

7.2.2 readers.buffer

The [*readers.buffer*](#) (page 51) 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 355) 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 355) for an example usage scenario for [*readers.buffer*](#) (page 51).

Options

7.2.3 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

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

Options

bounds What spatial extent should points be generated within? Text string of the form “[xmin,xmax],[ymin,ymax],[zmin,zmax]”. [Default: unit cube]

count How many synthetic points to generate before finishing? [Required]

mean_xlylz Mean value in the x, y, or z dimension respectively. (Normal mode only) [Default: 0]

stdev_xlylz Standard deviation in the x, y, or z dimension respectively. (Normal mode only) [Default: 1]

mode “constant”, “random”, “ramp”, “uniform”, “normal” or “grid” [Required]

7.2.4 readers.gdal

The [GDAL](http://gdal.org) (<http://gdal.org>) reader reads [GDAL](http://www.gdal.org/formats_list.html) 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”. The user must know what the bands correspond to, and use [filters.ferry](#) (page 125) to copy data into known [Dimensions](#) (page 185) as needed.

Note: [filters.ferry](#) (page 125) is needed to map GDAL output to typical [Dimensions](#) (page 185) names. For output to formats such as [LAS](#) (page 85), this mapping is required.

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.

```
1  {
2      "pipeline": [
3          {
4              "type": "readers.gdal",
5              "filename": "./pdal/test/data/autzen/autzen.jpg"
6          },
7          {
8              "type": "writers.text",
9              "filename": "outputfile.txt"
10         }
11     ]
12 }
```

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) (<http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html>) file using [writers.las](#) (page 85).

```
1 {
2     "pipeline": [
3         {
4             "type": "readers.gdal",
5             "filename": "./pdal/test/data/autzen/autzen.jpg"
6         },
7         {
8             "type": "filters.ferry",
9             "dimensions": "band-1=Red, band-2=Green, band-3=Blue",
10            },
11            {
12                "type": "writers.text",
13                "filename": "outputfile.txt"
14            }
15        ]
16 }
```

Options

filename [GDALOpen](#)

(http://www.gdal.org/gdal_8h.html#a6836f0f810396c5e45622c8ef94624d4) ‘able raster file to read [Required]

count Maximum number of points to read [Optional]

7.2.5 readers.geowave

The **GeoWave reader** uses [GeoWave](https://ngageoint.github.io/geowave/) (<https://ngageoint.github.io/geowave/>) to read from Accumulo. GeoWave entries are stored using [EPSG:4326](http://epsg.io/4326/) (<http://epsg.io/4326/>). Instructions for configuring the GeoWave plugin can be found [here](https://ngageoint.github.io/geowave/documentation.html#jace-jni-proxies-2) (<https://ngageoint.github.io/geowave/documentation.html#jace-jni-proxies-2>).

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```
{
  "pipeline": [
    {
      "type": "readers.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",
      "bounds": "[[0,1000000], [0,1000000], [0,100]]",
      "filename": "./pdal/test/data/autzen/autzen.jpg"
    },
    {
      "type": "writers.text",
      "filename": "outputfile.txt"
    }
  ]
}
```

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]

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]

7.2.6 readers.greyhound

The **Greyhound Reader** allows you to query point data from a Greyhound (<https://github.com/hobu/greyhound>) server.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```
{  
  "pipeline": [  
    {  
      "type": "readers.greyhound",  
      "url": "data.greyhound.io",  
      "resource": "iowa-z",  
      "filter": {"$and": [  
        {"Path": "02004736.laz"},  
        {"Classification": {"$ne": 7}}  
      ]}  
    },  
    "output.laz"  
  ]  
}
```

```
{  
  "pipeline": [  
    "greyhound://data.greyhound.io/resource/iowa-z/read?filter={\\"  
    \+Path\":\\"02004736.laz\\\"},  
    "02004736.laz"  
  ]  
}
```

Options

url Greyhound server URL, or a full Greyhound [read](#) (<https://github.com/hobu/greyhound/blob/master/doc/clientDevelopment.rst#the-read>-

query) query URL. If specified as a full Greyhound query URL, no other options need to be present.

resource Name of the Greyhound resource to access.

bounds Spatial bounds to query, expressed as a string, e.g. $[x_{min}, x_{max}], [y_{min}, y_{max}]$ or $[x_{min}, x_{max}], [y_{min}, y_{max}], [z_{min}, z_{max}]$ or as a Greyhound [bounds array](#) (<https://github.com/hobu/greyhound/blob/master/doc/clientDevelopment.rst#bounds-option>). By default, the entire resource is queried.

depth_begin Beginning octree depth to query, inclusive. Lower depth values have coarser resolution, so a depth range of $[0, 8]$ could provide a low-resolution overview of the entire resource, for example. [Default: **0**]

depth_end Ending octree depth to query, non-inclusive. A value of **0** will search all depths greater-than or equal-to *depth_begin*. If non-zero, this value should be greater than *depth_begin* or the result will always be empty. [Default: **0**]

tile_path A Greyhound resource may be an aggregation of multiple input files. If a *tile_path* option is present, then only points belonging to that file will be queried. This search is spatially optimized, so no [bounds](#) (page 57) option needs to be present to limit the query bounds.

filter Server-side filtering may be requested which may further limit the data selected by the query. The filter is represented as JSON, and performs filtering on dimensions present in the resource, or the pseudo-dimension *Path*, corresponding to [tile_path](#) (page 57) values.

Arbitrary logic combinations may be created using [comparison](#) (<https://docs.mongodb.com/manual/reference/operator/query-comparison/>) and [logical](#) (<https://docs.mongodb.com/manual/reference/operator/query-logical/>) query operators with syntax matching that of MongoDB. Some sample filters follow.

```
{  
  "Path": {"$in": ["tile-845.laz", "tile-846.laz"]},  
  "Classification": {"$ne": 18}  
}
```

```
{"$or": [  
  {"Red": {"$gt": 200}},  
  {"Blue": {"$gt": 120, "$lt": 130}},  
  {"Classification": {"$nin": [2, 3]}}  
]
```

7.2.7 readers.ilvis2

The **ILVIS2 reader** read from files in the ILVIS2 format. See <http://nsidc.org/data/docs/daac/icebridge/ilvis2/index.html> for more information

Parameter Description

The IceBridge LVIS Level-2 Geolocated Surface Elevation Product ASCII text format data files contain fields as described in Table 2.

Table 2. ASCII Text File Parameter Description

Parameter	Description	Units
LVIS_LFID	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.	n/a
SHOTNUMBER	Laser shot assigned during collection	n/a
TIME	UTC decimal seconds of the day	Seconds
LONGITUDE_CENTROID	Refers to the centroid longitude of the corresponding LVIS Level-1B waveform.	Degrees east
LATITUDE_CENTROID	Refers to the centroid latitude of the corresponding LVIS Level-1B waveform.	Degrees north
ELEVATION_CENTROID	Refers to the centroid elevation of the corresponding LVIS Level-1B waveform.	Meters
LONGITUDE_LOW	Longitude of the lowest detected mode within the waveform	Degrees east
LATITUDE_LOW	Latitude of the lowest detected mode within the waveform	Degrees north
ELEVATION_LOW	Mean elevation of the lowest detected mode within the waveform	Meters
LONGITUDE_HIGH	Longitude of the center of the highest mode in the waveform	Degrees east
LATITUDE_HIGH	Latitude of the center of the highest mode in the waveform	Degrees north
ELEVATION_HIGH	Elevation of the center of the highest mode in the waveform	Meters

Fig. 7.3: Dimensions provided by the ILVIS2 reader

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations

Example

```
{  
  "pipeline": [  
    {  
      "type": "readers.ilvis2",  
      "files": [  
        "file1.las",  
        "file2.las"  
      ]  
    }  
  ]  
}
```

```

    "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]

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

metadata XML metadata file to coincidentally read [Optional]

count Maximum number of points to read [Optional]

7.2.8 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://www.laszip.org>) files. In order to use compressed LAS, your version of PDAL must be built with one of the two supported decompressors, [LASzip](#) (<http://www.laszip.org>) or [LAZperf](#) (<https://github.com/verma/laz-perf>). See the [compression](#) (page 61) 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 85) 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](#) (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

```
1  {
2      "pipeline": [
3          {
4              "type": "readers.las",
5              "filename": "inputfile.las"
6          },
7          {
8              "type": "writers.text",
9              "filename": "outputfile.txt",
10         }
11     ]
12 }
```

Options

filename LAS file to read [Required]

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>, ... where type is one of: int8, int16, int32, int64, uint8, uint16, uint32, uint64, float, double. _t may be added to any of the type names as well (e.g., uint32_t).

Note: The presence of an extra bytes VLR causes when reading a version 1.4 LAS file causes this option to be ignored.

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: “laszip”]

spatialreference Sets the spatial reference for the file data. Overrides any spatial reference information in the file itself. Most text-based formats of SRS information are accepted, including WKT and proj.4.

count Maximum number of points read [Optional]

7.2.9 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 *Dimensions* (page 185) 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

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

Options

filename Output file name [REQUIRED]

struct Array structure name to read [OPTIONAL, defaults PDAL]

7.2.10 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](#) (<http://www.ldeo.columbia.edu/res/pi/MB-System/>) library to read the data and therefore supports [all formats](#) (<https://www.ldeo.columbia.edu/res/pi/MB-System/html/mbio.html#lbAI>) 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.

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

Options

filename Filename to read from [Required]

format Name of number of format of file being read. See MB-System documentation for a list of all formats (<https://www.ledo.columbia.edu/res/pi/MB-System/html/mbio.html#lbAI>). [Required]

count Maximum number of points to read [Optional]

7.2.11 readers.mrsid

Implements MrSID 4.0 LiDAR Compressor. It requires the [Lidar_DSDK](#) (<https://www.lizardtech.com/developer/>) to be able to decompress and read data.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

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

Options

filename Filename to read from [Required]

7.2.12 readers.nitf

The [NITF](http://en.wikipedia.org/wiki/National_Imagery_Transmission_Format) (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) (<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://jitic.fhu.disa.mil/cgi/nitf/registers/desreg.aspx) (<http://jitic.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 59) and *writers.las* (page 85) stages to actually read and write the data.

Note: PDAL uses a fork of the [NITF Nitro](http://nitro-nitf.sourceforge.net/wikka.php?wakka=HomePage) (<http://nitro-nitf.sourceforge.net/wikka.php?wakka=HomePage>) library available at

<https://github.com/hobu/nitro> for NITF read and write support.

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations

Example

```
1  {
2      "pipeline": [
3          {
4              "type": "readers.nitf",
5              "filename": "mynitf.nitf"
6          },
7          {
8              "type": "writers.las",
9              "filename": "outputfile.las"
10         }
11     ]
12 }
```

Options

filename Filename to read from [Required]

count Maximum number of points to read [Optional]

spatialreference Spatial reference to apply to data

extra_dims Dimensions to assign to extra byte data

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: “laszip”]

7.2.13 readers.oci

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

```
{  
  "pipeline": [  
    {  
      "type": "readers.oci",  
      "query": "SELECT \r\n          1.\"OBJ_ID\", 1.\"BLK_ID\", 1.\\"  
      ↵ \"BLK_EXTENT\", \r\n          1.\"BLK_DOMAIN\", 1.\"PCBLK_MIN_RES\"  
      ↵ \", \r\n          1.\"PCBLK_MAX_RES\", 1.\"NUM_POINTS\", \r\n      ↵ \r\n          1.\"NUM_UNSORTED_POINTS\", 1.\"PT_SORT_DIM\", \r\n      ↵ \r\n          1.\"POINTS\", b.cloud\r\n          FROM AUTZEN_BLOCKS 1, AUTZEN_  
      ↵ CLOUD b\r\n          WHERE 1.obj_id = b.id and 1.obj_id in (1,  
      ↵ 2)\r\n          ORDER BY 1.obj_id",  
      "connection": "grid/grid@localhost/orcl",  
      "populate_pointsourceid": "true"  
    },  
    {  
      "type": "writers.las",  
      "filename": "outputfile.las"  
    }  
  ]  
}
```

Options

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]

spatialreference

spatialreference Sets the spatial reference for the point ata. Overrides any spatial
reference information read from the database. Most text-based formats of SRS

information are accepted, including WKT and proj.4.

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**]

7.2.14 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

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

Options

filename csd file to read [Required]

count Maximum number of points read [Optional]

7.2.15 readers.pcd

The **PCD Reader** supports reading from [Point Cloud Data \(PCD\)](#) (http://pointclouds.org/documentation/tutorials/pcd_file_format.php) formatted files, which are

used by the Point Cloud Library (PCL) (<http://pointclouds.org>).

Note: The *PCD Reader* requires linkage of the [PCL](http://pointclouds.org) (<http://pointclouds.org>) library.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```
{  
  "pipeline": [  
    {  
      "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 [Optional]

7.2.16 readers.pgPointCloud

The **PostgreSQL Pointcloud Reader** allows you to read from a PostgreSQL database that the [PostgreSQL Pointcloud](https://github.com/pramsey/pgPointCloud) (<https://github.com/pramsey/pgPointCloud>) 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 on the way with a “where” clause.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```
{  
  "pipeline": [  
    {  
      "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

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**]

spatialreference Sets the spatial reference for the point data. Overrides any spatial reference information read from the database. Most text-based formats of SRS information are accepted, including WKT and proj.4.

count Maximum number of points to read [Optional]

7.2.17 readers.ply

The **ply reader** reads points and vertices from the [polygon file format](http://paulbourke.net/dataformats/ply/) (<http://paulbourke.net/dataformats/ply/>), a common file format for storing three dimensional

models. The [rply library](http://w3.impa.br/~diego/software/rply/) (<http://w3.impa.br/~diego/software/rply/>) is included with the PDAL source, so there are no external dependencies.

Note: 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

```
{  
  "pipeline": [  
    {  
      "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 [Optional]

7.2.18 readers pts

The **PTS reader** reads data from Leica Cyclone PTS files. It is not very sophisticated.

Default Embedded Stage

This stage is enabled by default

Example Pipeline

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

Options

filename text file to read [Required]

count Maximum number of points to read [Optional]

7.2.19 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

```
{  
  "pipeline": [  
    {  
      "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]

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.

count Maximum number of points to read [Optional]

7.2.20 readers.rxp

The **RXP reader** read from files in the RXP format, the in-house streaming format used by **RIEGL Laser Measurement Systems** (<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

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/) (<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](http://www.pdal.io/compilation/unix.html#configure-your-optional-libraries) (<http://www.pdal.io/compilation/unix.html#configure-your-optional-libraries>).

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.

```
{  
  "pipeline": [  
    {  
      "type": "readers.rxp",  
      "filename": "120304_204030.rxp",  
      "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 85). 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 164) to filter those points earlier in the pipeline.

Options

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

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]

minimal If “true”, only write X, Y, Z, and time values to the data stream. If “false”, write all available values as derived from the rxp file. Use this feature to reduce the memory footprint of a PDAL run, if you don’t need any values but the points themselves. [default: false]

reflectance_as_intensity If “true”, maps reflectance values onto intensity values using a range from -25dB to 5dB. [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]

7.2.21 readers.sbet

The **SBET reader** read from files in the SBET format, used for exchange data from interital measurement units (IMUs).

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations

Example

```
{  
  "pipeline": [  
    "sbetfile.sbet",  
    "output.las"  
  ]  
}
```

Options

filename File to read from [Required]

count Maximum number of points to read [Optional]

7.2.22 readers.sqlite

The [SQLite](https://sqlite.org/) (<https://sqlite.org/>) point cloud reader allows you to read data stored in a SQLite database using a scheme that PDAL wrote using the *writers.sqlite* (page 101) writer. Much like the *writers.oci* (page 93) and *writers.pgpointcloud* (page 98), 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

```
{
  "pipeline": [
    {
      "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\r\n                           FROM
      ↪simple_blocks l, simple_cloud b\r\n WHERE l.
      ↪cloud = b.cloud and l.cloud in (1)\r\n                           order by
      ↪l.cloud"
    },
    {
      "type": "writers.las",
      "filename": "outputfile.las"
    }
  ]
}
```

Options

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

spatialreference The spatial reference to use for the points. Over-rides the value read from the database.

count Maximum number of points to read [Optional]

7.2.23 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) (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 1) indicate the separator character for the fields and 2) name the dimension for each field in the points. Any single non-alphanumeric character can be used as a separator. The header line separator can be overridden by the ‘separator’ option (see below). 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.

Blank lines after the header line are ignored.

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
289817.53, 4320980.16, 170.62
289818.01, 4320980.38, 170.61
289818.50, 4320980.59, 170.58
```

Example Pipeline

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

Options

filename text file to read [Required]

separator Separator character to override that found in header line.

count Maximum number of points to read [Optional]

7.2.24 readers.tindex

A [GDAL tile index](http://www.gdal.org/gdaltindex.html) (<http://www.gdal.org/gdaltindex.html>) is an [OGR](http://gdal.org/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 34) application to generate tile index files in any format that [OGR](http://gdal.org/ogr/) (<http://gdal.org/ogr/>) supports writing. Once you have the tile index, you can then use the *readers.tindex* (page 77) driver 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 41) example to read and automatically merge the data.

```
{
  "pipeline": [
    {
      "type": "readers.tindex",
      "filter_srs": "+proj=lcc +lat_1=43 +lat_2=45.5 +lat_0=41.75
      ↳+lon_0=-120.5 +x_0=399999.999999999 +y_0=0 +ellps=GRS80 +units=ft
      ↳+no_defs",
      "filename": "index.sqlite",
      "where": "location LIKE '%interesting.las%'",
      "polygon": "POLYGON ((635629.85000000 848999.70000000, 635629.
      ↳85000000 853535.43000000, 638982.55000000 853535.43000000, 638982.
      ↳55000000 848999.70000000, 635629.85000000 848999.70000000))"
    },
    {
      "type": "writers.las",
      "filename": "outputfile.las"
    }
  ]
}
```

Options

filename OGROpen'able raster file to read [Required]

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

srs_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) (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 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.

filter_srs Transforms any wkt or boundary option to this coordinate system before filtering or reading data.

where [OGR SQL](http://www.gdal.org/ogr_sql.html) (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`

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

count Maximum number of points to read [Optional]

7.3 Writers

Writers consume data provided by *Readers* (page 50). 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 185)

7.3.1 writers.bpf

BPF is an NGA specification for point cloud data. The specification can be found at <https://nsgreg.nga.mil/doc/view?i=4202> 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

```
{  
  "pipeline": [  
    {  
      "type": "readers.bpf"  
      "filename": "inputfile.las"  
    },  
    {  
    }
```

```
        "type": "writers.bpf",
        "filename": "outputfile.bpf"
    }
]
}
```

Options

filename BPF file to read. 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 178), *filters.chipper* (page 106) or *filters.divider* (page 120). [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. NOTE: Only the UTM coordinate type is currently supported. [Default: 0, with coordinate type set to 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: 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.gdal

The [GDAL](http://gdal.org) (<http://gdal.org>) writer creates a raster from a point cloud using an interpolation algorithm. Output is produced using GDAL and can use any [driver that supports creation of rasters](#) (http://www.gdal.org/formats_list.html). A [data_type](#) (page 83) 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 83) 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 83) and the square root of two. This is consistent with the original [Points2Grid](http://www.opentopography.org/otsoftware/points2grid) (<http://www.opentopography.org/otsoftware/points2grid>) application from which this algorithm has its roots. 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 83) 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) (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 83) 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 83) 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.

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

Options

filename Name of output file. [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: *resolution* (page 83) * sqrt(2)]

gdaldriver Name of the GDAL driver 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 to use for the output raster (double, float, int32, uint16, etc.). Many GDAL drivers only support a limited set of output data types. The default value 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 83). -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]).

Note: The *bounds* (page 83) option is required when a pipeline is run in streaming mode.

7.3.3 writers.geowave

The **GeoWave writer** uses [GeoWave](https://ngageoint.github.io/geowave/) (<https://ngageoint.github.io/geowave/>) to write to Accumulo. GeoWave entries are stored using [EPSG:4326](http://epsg.io/4326) (<http://epsg.io/4326/>). Instructions for configuring the GeoWave plugin can be found [here](https://ngageoint.github.io/geowave/documentation.html#jace-jni-proxies-2) (<https://ngageoint.github.io/geowave/documentation.html#jace-jni-proxies-2>).

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```
{  
  "pipeline": [  
    {  
      "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 with 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]

7.3.4 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` 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` items.

```
{  
  "pipeline": [  
    {  
      "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",
            "data": "dGhpcyBpcyBzb21lIG1vcmUgdGV4dA=="
        }
    ],
    "filename": "outputfile.las"
}
]
}
```

Example

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

Options

filename LAS file to read. 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 178), [*filters.chipper*](#) (page 106) or [*filters.divider*](#) (page 120). [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, dataformat_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 1 and 65535) [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>, ...` where type is one of: int8, int16, int32, int64, uint8, uint16, uint32, uint64, float, double _t may be added to any of the type names as well (e.g., `uint32_t`). When the version of the output file is specified as 1.4 or greater, an extra bytes VLR (User ID: LASF_Spec, Record ID: 4), is created that describes the extra dimensions specified by this option.

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.

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/) (<http://www.json.org/>) output with both the *Metadata* (page 353) and *Pipeline* (page 41) serialization. [Default: **false**]

7.3.5 writers.matlab

The **Matlab Writer** supports writing Matlab *.mat* files.

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

Variables - PDAL	
PDAL	x
1x1 struct with 16 fields	
Field ▲	Value
X	1065x1 double
Y	1065x1 double
Z	1065x1 double
Intensity	1065x1 uint16
ReturnNumber	1065x1 uint8
NumberOfReturns	1065x1 uint8
ScanDirectionFlag	1065x1 uint8
EdgeOfFlightLine	1065x1 uint8
Classification	1065x1 uint8
ScanAngleRank	1065x1 single
UserData	1065x1 uint8
PointSourceId	1065x1 uint16
GpsTime	1065x1 double
Red	1065x1 uint16
Green	1065x1 uint16
Blue	1065x1 uint16

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

```
{  
  "pipeline": [  
    {  
      "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 Dimensions to include in the output file [OPTIONAL, defaults to all available dimensions]

struct Array structure name to read [OPTIONAL, defaults PDAL]

7.3.6 writers.nitf

The [NITF](http://en.wikipedia.org/wiki/National_Imagery_Transmission_Format) (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

```
{  
  "pipeline": [  
    {  
      "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

```
{  
  "pipeline": [  
    {  
      "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 178), [*filters.chipper*](#) (page 106) or [*filters.divider*](#) (page 120).

level 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]

7.3.7 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

```
{  
  "pipeline": [  
    {  
      "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.

7.3.8 writers.oci

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

```
{  
  "pipeline": [  
    {  
      "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 outputting 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**]

7.3.9 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

```
{  
  "pipeline": [  
    "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 178), [filters.chipper](#) (page 106) or [filters.divider](#) (page 120).

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’.

7.3.10 writers.pcd

The **PCD Writer** supports writing to [Point Cloud Data \(PCD\)](#) (http://pointclouds.org/documentation/tutorials/pcd_file_format.php) 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. When compression is enabled, the output is PCD’s binary-compressed format.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Note: The *PCD Writer* requires linkage of the [PCL](#) (<http://pointclouds.org>) library.

Example

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

Options

filename PCD file to write [Required]

compression Apply compression to the PCD file? [Default: false]

7.3.11 writers.pgPointCloud

The **PostgreSQL Pointcloud Writer** allows you to write to PostgreSQL database that have the [PostgreSQL Pointcloud](http://github.com/pramsey/pgPointCloud) (<http://github.com/pramsey/pgPointCloud>) 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 106).

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```
{
  "pipeline": [
    {
      "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
- **ght** applies a “geohash tree” compression by sorting the points into a prefix tree

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.

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.

7.3.12 writers.ply

The **ply writer** writes the [polygon file format](http://paulbourke.net/dataformats/ply/) (<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 127) and [filters.poisson](#) (page 160) create a mesh suitable for output as faces.

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations

Example

```
{  
  "pipeline": [  
    {  
      "type": "readers.pcd",  
      "name": "myreader"  
    },  
    {  
      "type": "writers.ply",  
      "name": "myply"  
    }  
  ]  
}
```

```

        "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’, and ‘default’. ‘default’ is binary output in the endianness of the machine. [Default: ‘default’]

dims List of dimensions to write as elements. [Default: all dimensions]

faces Write a mesh as faces in addition to writing points as vertices. [Default: false]

7.3.13 writers.sqlite

The [SQLite](http://sqlite.org) (<http://sqlite.org>) driver outputs point cloud data into a PDAL-specific scheme that matches the approach of *readers.pgpointcloud* (page 68) and *readers.oci* (page 66).

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```

{
    "pipeline": [
        {
            "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",
        "cloud_column_name": "CLOUD",
        "filename": "outputfile.pcd"
    }
]
}
```

Options

connection SQLite 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 <https://github.com/verma/laz-perf> compression technique to store patches

overwrite To drop the table before writing set to ‘true’. 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.

7.3.14 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) (<http://geojson.org>) and [CSV](http://en.wikipedia.org/wiki/Comma-separated_values) (http://en.wikipedia.org/wiki/Comma-separated_values) output.

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations

Example

```
{
  "pipeline": [
    {
      "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**]

order Comma-separated list of dimension names, giving the desired column order in the output file, for example “X,Y,Z,Red,Green,Blue”. [Default: none]

keep_unspecified Should we output any fields that are not specified in the dimension order? [Default: **true**]

jcallback 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`]

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: `,`]

7.4 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 173) 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 filters.approximatecoplanar

`filters.approximatecoplanar` filter estimates the planarity of a neighborhood of points by first computing eigenvalues for the points and then tagging those points for which the following is true:

$$\lambda_1 > (thresh_1 * \lambda_0) \& \& (\lambda_1 * thresh_2) > \lambda_2$$

where λ_0 , λ_1 , λ_2 are the eigenvalues in ascending order. The threshold values `thresh1` and `thresh2` 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).

Eigenvalue estimation is performed using Eigen’s `SelfAdjointEigenSolver`. For more information see https://eigen.tuxfamily.org/dox/classEigen_1_1SelfAdjointEigenSolver.html.

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 `filters.approximatecoplanar` filter. A

`filters.range` stage then filters out any points that were not deemed to be coplanar before writing the result in compressed LAZ.

```
{  
  "pipeline": [  
    "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**]

7.4.2 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

Example 1

This pipeline resets the Classification of all points with classifications 2 or 3 to 0 and all points with classification of 5 to 4.

```
{  
  "pipeline": [  
    "autzen-dd.las",  
    {  
      "type": "filters.assign",  
      "assignment" : "Classification[2:3]=0",  
      "assignment" : "Classification[5:5]=4"  
    },  
    {  
      "filename": "attributed.las",  
      "scale_x": 0.0000001,  
      "scale_y": 0.0000001  
    }  
  ]  
}
```

Options

assignment A [range](#) (page 172) 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.

7.4.3 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 a an irregular tiling of the input data.

Note: Each chip will have approximately, but not exactly, the `capacity` point count specified.

See also:

The [split](#) (page 33) utilizes the [filters.chipper](#) (page 106) 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

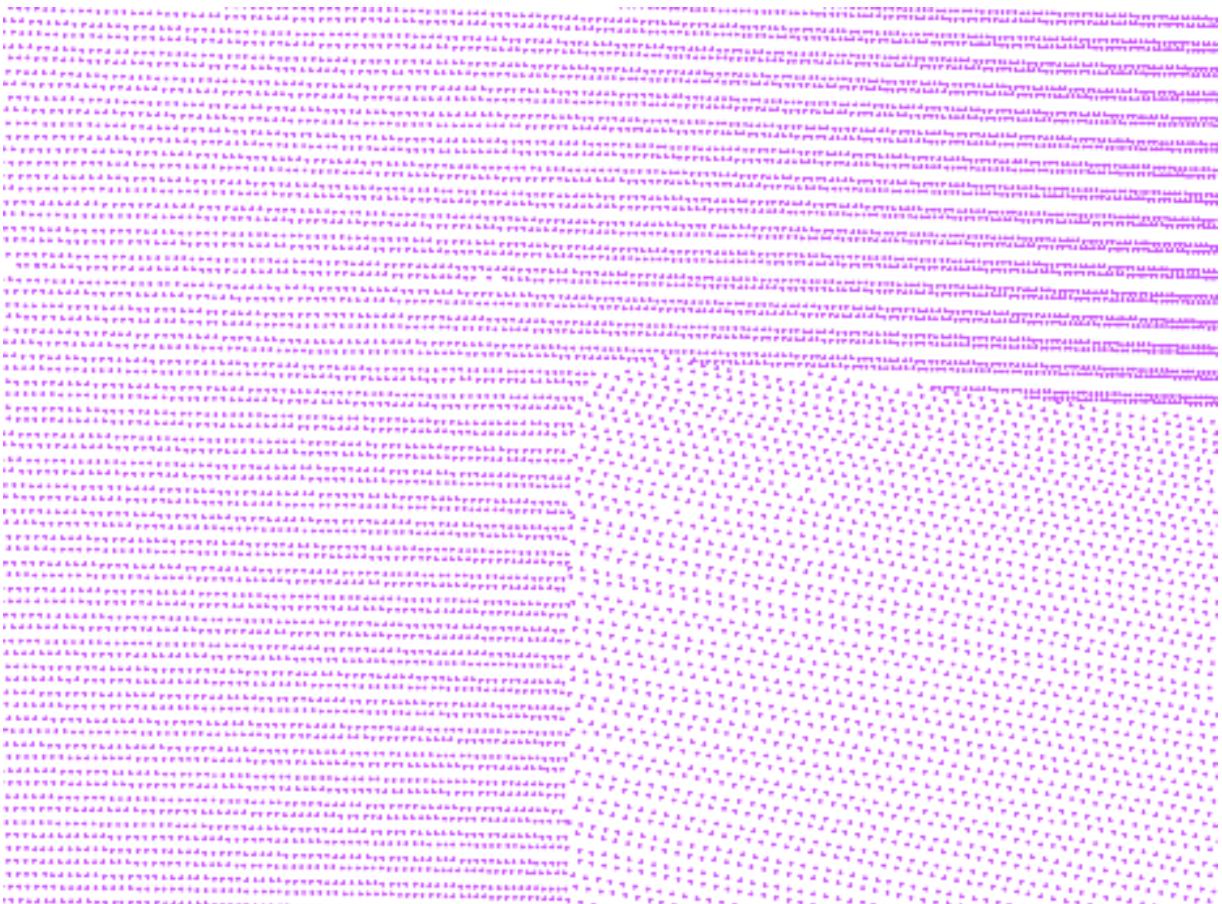


Fig. 7.4: Before chipping, the points are all in one collection.

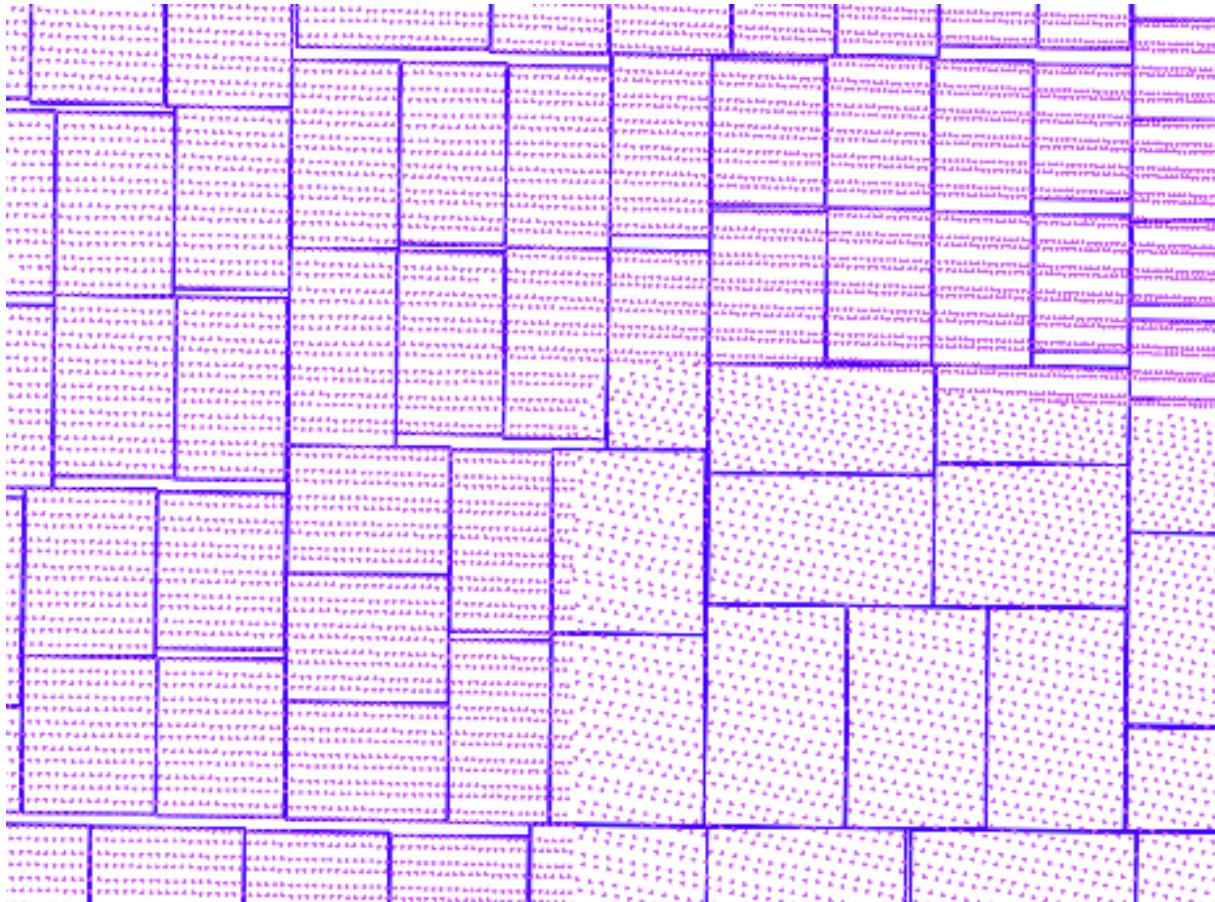


Fig. 7.5: After chipping, the points are tiled into smaller contiguous chips.

Example

```
{
  "pipeline": [
    "example.las",
    {
      "type": "filters.chipper",
      "capacity": "400",
    },
    {
      "type": "writers.pgpointcloud",
      "connection": "dbname='lidar' user='user'"
    }
  ]
}
```

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**]

7.4.4 filters.cluster

The Cluster filter first performs Euclidean Cluster Extraction on the input PointView and then labels each point with its associated cluster ID.

Default Embedded Stage

This stage is enabled by default

Example

```
{
  "pipeline": [
    "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: **UINT64_MAX**]

tolerance Cluster tolerance - maximum Euclidean distance for a point to be added to the cluster. [Default: **1.0**]

7.4.5 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` and `maximum`, and the data are scaled for the given dimension accordingly.
2. You provide a `k` and a `mad` setting, and the scaling is set based on Median Absolute Deviation.
3. You provide a `k` setting and the scaling is set based on the `k`-number of standard deviations from the median.

You can provide your own [GDAL](http://www.gdal.org) (<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 111) for more information.

Note: [filters.colorinterp](#) (page 110) will use the entire band to scale the colors.

Default Embedded Stage

This stage is enabled by default

Example

```
{  
  "pipeline": [  
    "uncolored.las",  
    {  
      "type": "filters.colorinterp",  
      "ramp": "pestel_shades",  
      "mad": true,  
      "k": 1.8,  
      "dimension": "Z"  
    },  
    "colorized.las"  
  ]  
}
```

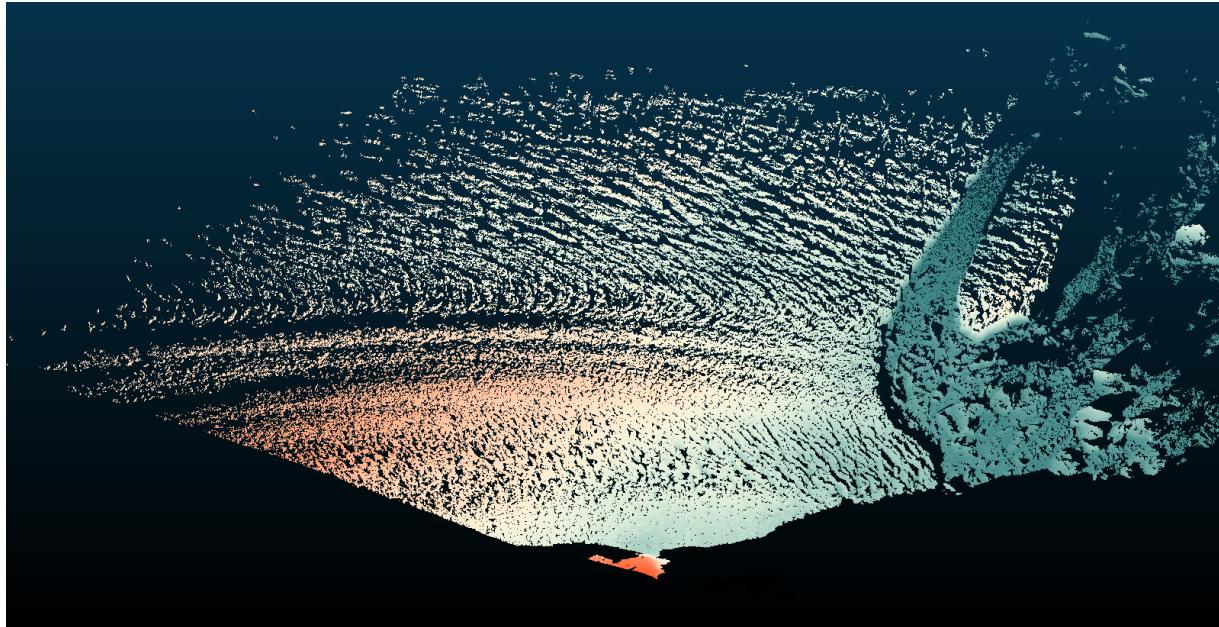


Fig. 7.6: Image data with interpolated colors based on Z dimension and pestel_shades ramp.

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` option to the filter and it will be used. Otherwise, provide a [GDAL](http://www.gdal.org) (<http://www.gdal.org>)-readable raster filename.

`awesome_green`

`black_orange`

`blue_orange`

`blue_hue`

`blue_orange`

`blue_red`

`heat_map`

`pestel_shades`

Options

ramp The raster file to use for the color ramp. Any format supported by [GDAL](http://www.gdal.org) (<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` value is also provided, the `k` 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` value is also provided, the `k` 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` and `maximum` will be computed from the median and setting them will have no effect.

mad If true, `minimum` and `maximum` will be computed by the median absolute deviation. See [filters.mad](#) (page 143) for discussion. [Default: `false`]

mad_multiplier MAD threshold multiplier. Used in conjunction with `k` to threshold the differencing. [Default: 1.4862]

7.4.6 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.

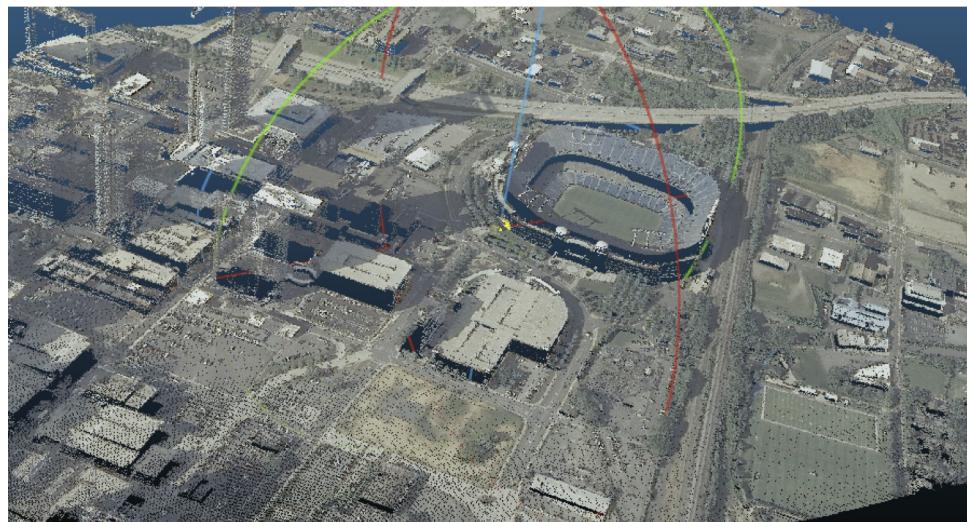


Fig. 7.7: After colorization, points take on the colors provided by the input image

Note: [GDAL](http://www.gdal.org) (<http://www.gdal.org>) is used to read the color information and any GDAL-readable [supported format](#) 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

```
{  
  "pipeline": [  
    "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_CACEMAX` 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
```

```
{  
  "pipeline": [  
    "colourless.laz",  
    {  
      "type": "filters.colorization",  
    }  
  ]  
}
```

```

    "raster": "orth-striped.tif"
},
"coloured-striped.las"
]
}
}
```

Simple application of the [filters.colorization](#) (page 113) using the striped TIFF (http://www.gdal.org/frmt_gtiff.html) with a 268mb [readers.las](#) (page 59) 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_CACEMAX` variable to a size larger than the TIFF file dramatically speeds up the color fetching:

```
[hobu@pyro knudsen (master)]$ export GDAL_CACEMAX=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 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. 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”]

7.4.7 filters.computerange

The Compute Range filter computes the range from the sensor to each of the detected returns.

Note: The Compute Range filter is specific to raw data from a particular data provider, where the sensor coordinates for each frame are encoded as regular points, and are identified by the pixel number -5.

Default Embedded Stage

This stage is enabled by default

Example

```
{  
  "pipeline": [  
    "input.bpf",  
    {  
      "type": "filters.computerange"  
    },  
    {  
      "type": "writers.bpf",  
      "filename": "output.bpf",  
      "output_dims": "X, Y, Z, Range"  
    }  
  ]  
}
```

7.4.8 filters.cpd

The CPD filter uses the Coherent Point Drift [\[MS10\]](#) (page 457) 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 [\[YG88\]](#) (page 457) 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.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Examples

```
{  
    "pipeline": [  
        "fixed.las",  
        "moving.las",  
        {  
            "type": "filters.cpd",  
            "method": "rigid"  
        },  
        "output.las"  
    ]  
}
```

If “method” 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:

```
$ pdal pipeline cpd-pipeline.json --metadata cpd-metadata.json
```

The metadata output might start something like:

```
{  
    "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-18 -3.49247e-10 -2.  
-1684e-19 1.73472e-18 1 -1.53477e-12 0  
0 0 1"  
                },  
            ],  
        }  
    ]  
}
```

See also:

filters.transformation (page 180) to apply a transform to other points.

Options

method Change detection method to use. Valid values are “rigid”, “affine”, and “nonrigid”.
[Default: **rigid**]

7.4.9 filters.crop

The crop filter removes points that fall outside or inside a cropping bounding box (2D), 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* 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

```
{  
  "pipeline": [  
    "file-input.las",  
    {  
      "type": "filters.crop",  
      "bounds": "[0,1000000], [0,1000000]"  
    },  
    {  
      "type": "writers.las",  
      "filename": "file-cropped.las"  
    }  
  ]  
}
```

Options

bounds The extent of the clipping rectangle, expressed in a string, eg: $([xmin, xmax], [ymin, ymax])$ This option can be specified more than once.

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.

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. Requires **distance**.

distance Distance in units of common X, Y, and Z *Dimensions* (page 185) to crop circle or sphere in combination with **point**.

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 (if possible).

7.4.10 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

Example

```
{  
  "pipeline": [  
    {  
      "type": "readers.las",  
      "filename": "larger.las"  
    },  
    {  
      "type": "filters.decimation",  
      "step": 10  
    },  
    {
```

```
        "type": "writers.las",
        "filename": "smaller.las"
    }
]
}
```

See also:

[filters.voxelgrid](#) (page 184) 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]

7.4.11 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.

```
{
  "pipeline": [
    "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]

Warning: You must specify exactly one of either count or capacity.

7.4.12 filters.eigenvalues

`filters.eigenvalues` 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`. For more information see 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.

```
{  
  "pipeline": [  
    "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**]

7.4.13 filters.elm

The Extended Local Minimum (ELM) filter marks low points as noise. This filter is an implementation of the method described in [\[Chen2012\]](#) (page 457).

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.

```
{
  "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 value of 2.0, and finishes by extracting all returns that are not marked as noise.

```
{
  "pipeline": [
    "input.las",
    {
      "type": "filters.assign",
      "assignment": "Classification[:] = 0"
    },
    {
      "type": "filters.elm",
      "threshold": 2.0
    },
    {
      "type": "filters.range",
      "limits": "Classification![7:7]"
    },
    "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**]

7.4.14 filters.estimaterank

`filters.estimaterank` computes the rank (i.e., the number of nonzero singular values) of a neighborhood of points.

This method uses Eigen's JacobiSVD class to solve the singular value decomposition and to estimate the rank using the user-specified threshold. A singular value will be considered nonzero if its absolute value is greater than the product of the user-supplied threshold and the absolute value of the maximum singular value.

More on JacobiSVD can be found at

https://eigen.tuxfamily.org/dox/classEigen_1_1JacobiSVD.html.

Default Embedded Stage

This stage is enabled by default

Example

This sample pipeline estimates the rank of each point using `filters.estimaterank` and then filters out those points where the rank is three using `filters.range`.

```
{  
  "pipeline": [  
    "input.las",  
    {  
      "type": "filters.estimaterank",  
      "knn": 8,  
      "thresh": 0.01  
    },  
    {  
      "type": "filters.range",  
      "limits": "Rank! [3:3]"  
    },  
    "output.laz"  
  ]  
}
```

Options

knn The number of k-nearest neighbors. [Default: **8**]

thresh The threshold used to identify nonzero singular values. [Default: **0.01**]

7.4.15 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.

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.

```
{  
  "pipeline": [  
    "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.

```
{  
    "pipeline": [  
        {  
            "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.

7.4.16 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

```
{
  "pipeline": [
    "input.las",
    {
      "type": "filters.greedyprojection",
      "multiplier": 2,
      "radius": 10
    },
    {
      "type": "writers.las",
      "filename": "output.las"
    }
  ]
}
```

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]

7.4.17 filters.gridprojection

The Grid Projection filter passes data through the Point Cloud Library ([PCL](#) (<http://www.pointclouds.org>)) GridProjection algorithm.

GridProjection is an implementation of the surface reconstruction method described in [[Li2010](#)] (page 457).

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```
{  
  "pipeline": [  
    "input.las",  
    {  
      "type": "filters.gridprojection"  
    },  
    {  
      "type": "writers.las",  
      "filename": "output.las"  
    }  
  ]  
}
```

Options

None at the moment. Relying on defaults within PCL.

7.4.18 filters.groupby

The groupby filter takes a single PointView as its input and creates a PointView for each category in the named dimension as its output.

Default Embedded Stage

This stage is enabled by default

Example

```
{  
  "pipeline": [  
    "input.las",  
    {  
      "type": "filters.groupby",  
      "dimension": "Classification"  
    },  
    "output_#.las"  
  ]  
}
```

Options

dimension The dimension containing data to be grouped.

7.4.19 filters.hag

The Height Above Ground (HAG) filter takes as input a point cloud with a Classification dimension, with ground points assigned the classification label of 2 (per LAS specification).

Note: We expect ground returns to have the classification value of 2 in keeping with the ASPRS Standard LIDAR Point Classes (see http://www.asprs.org/a/society/committees/standards/LAS_1_4_r13.pdf).

This could, for example, be generated by *filters.pmf* (page 159) or *filters.smrf* (page 176) (see *Identifying ground returns using ProgressiveMorphologicalFilter segmentation* (page 214)), but you can use whichever method you choose, as long as the ground returns are marked.

It returns a point cloud with a new dimension `HeightAboveGround` that contains the normalized height value.

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 HAG filter works by iterating through all points, finding the nearest neighbor (in XY only) amongst the ground points, and computing the distance between the two Z values.

The process of computing normalized heights is straightforward. First, we must have an estimate of the underlying terrain model. With this we can compute the difference between each point's elevation and the elevation of the terrain model at the same XY coordinate. The quality of the normalized heights will be a function of the quality of the terrain model, which of course depends on the quality of the ground segmentation approach and any interpolation that is required to arrive at the terrain elevation for a given XY coordinate. We will use a nearest neighbor interpolation scheme to estimate terrain elevations.

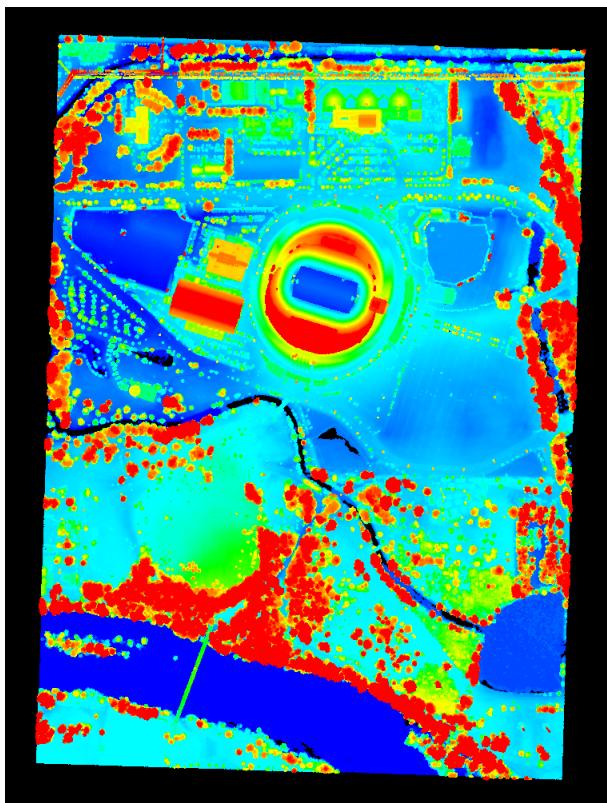
To compute the normalized heights, we first create a 2D KdTree (X and Y only) to accelerate our nearest neighbor search. The tree is composed of only ground returns. We then iterate over each of our points, searching for the nearest neighbor in the ground points. We then compute the difference between the elevation of the query point and the nearest neighbor in the ground set. This value is encoded as a new dimension called HeightAboveGround.

Default Embedded Stage

This stage is enabled by default

Example #1

Using the autzen dataset (here shown colored by elevation)



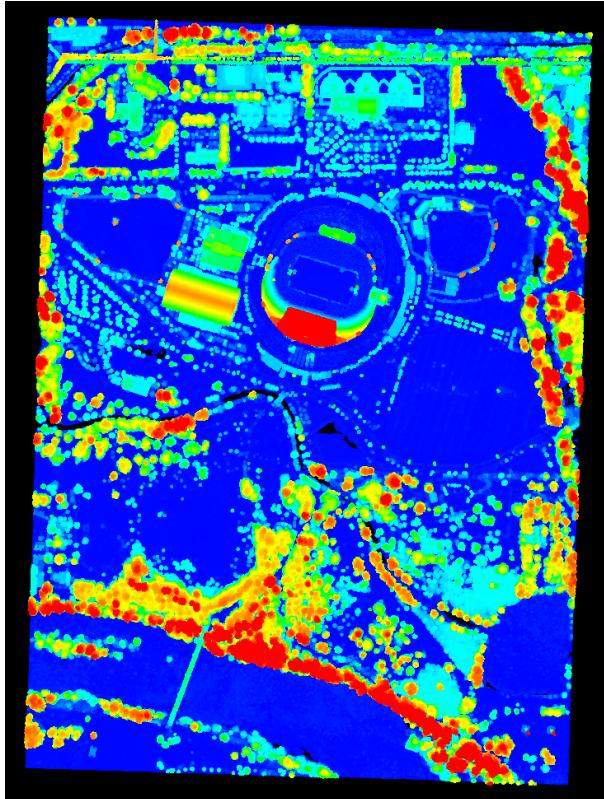
we execute the following pipeline

```
{  
  "pipeline": [  
    "autzen.laz",  
    {  
      "type": "filters.hag"  
    },  
    {  
      "type": "writers.bpf",  
      "filename": "autzen-height.bpf",  
      "output_dims": "X,Y,Z,HeightAboveGround"  
    }  
  ]  
}
```

which is equivalent to the pdal translate command

```
$ pdal translate autzen.laz autzen-height.bpf hag \  
  --writers.bpf.output_dims="X,Y,Z,HeightAboveGround"
```

In either case, the result, when colored by the normalized height instead of elevation is



Example #2

In the previous example, we chose a [writer](#) (page 79) that could output custom dimensions. If you'd instead like to overwrite your Z values, then follow the height filter with [filters.ferry](#) (page 125) as shown

```
{  
  "pipeline": [  
    "autzen.laz",  
    {  
      "type": "filters.hag"  
    },  
    {  
      "type": "filters.ferry",  
      "dimensions": "HeightAboveGround=Z"  
    },  
    "autzen-height-as-Z.laz"  
  ]  
}
```

which is equivalent to the command

```
$ pdal translate autzen.laz autzen-height-as-Z.laz hag ferry \  
  --filters.ferry.dimensions="HeightAboveGround=Z"
```

Example #3

If you don't yet have points classified as ground, start with [filters.pmf](#) (page 159) or [filters.smrf](#) (page 176) to label ground returns, as shown

```
{  
  "pipeline": [  
    "autzen.laz",  
    {  
      "type": "filters.smrf"  
    },  
    {  
      "type": "filters.hag"  
    },  
    {  
      "type": "filters.ferry",  
      "dimensions": "HeightAboveGround=Z"  
    },  
    "autzen-height-as-Z-smrf.laz"  
  ]  
}
```

which is once again equivalent to the command

```
$ pdal translate autzen.laz autzen-height-as-Z-smrf.bpf smrf hag_
↳ferry \
--filters.ferry.dimensions="HeightAboveGround=Z"
```

Options

None

7.4.20 filters.head

The HeadFilter returns a specified number of points from the beginning of the 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 170) and then picking the first 10000 using the HeadFilter.

```
{
  "pipeline": [
    {
      "type": "filters.randomize"
    },
    {
      "type": "filters.head",
      "count": 10000
    }
  ]
}
```

Example #2

Compute height above ground and extract the ten highest points.

```
{
  "pipeline": [
    {
      "type": "filters.smrf"
    },
    {
      "type": "filters.hag"
    },
    {
      "type": "filters.sort",
      "dimension": "HeightAboveGround",
      "order": "DESC"
    },
    {
      "type": "filters.head",
      "count": 10
    }
  ]
}
```

See also:

filters.tail (page 179) is the dual to *filters.head* (page 133).

Options

count Number of points to return. [Default: **10**]

7.4.21 filters.hexbin

A common question 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.

The hexbin filter reads a point stream and writes out a metadata record that contains a much tighter data bound, expressed as a well-known text polygon. In order to write out the metadata record, the *pdal* pipeline command must be invoked using the *-pipeline-serialization* option:

Dynamic Plugin



Fig. 7.8: Hexbin output shows boundary of actual points in point buffer, not just rectangular extents.

This stage requires a dynamic plugin to operate

Example

```
$ pdal pipeline hexbin-pipeline.json --pipeline-serialization hexbin-  
→out.json
```

After running with the pipeline serialization option, the output file looks like this:

```
{  
  "pipeline": [  
    {  
      "execution_metadata": {  
        "comp_spatialreference": "",  
        "compressed": false,  
        "count": 1065,  
        "creation_doy": 0,  
        "creation_year": 0,  
        "dataformat_id": 3,  
        "dataoffset": 229,  
        "filesource_id": 0,  
        "global_encoding": 0,  
        "global_encoding_base64": "AAA=",
```

```

"header_size": 227,
"major_version": 1,
"maxx": 638982.55,
"maxy": 853535.43,
"maxz": 586.38,
"minor_version": 2,
"minx": 635619.85,
"miny": 848899.7,
"minz": 406.59,
"offset_x": 0,
"offset_y": 0,
"offset_z": 0,
"project_id": "00000000-0000-0000-0000-000000000000",
"scale_x": 0.01,
"scale_y": 0.01,
"scale_z": 0.01,
"software_id": "TerraScan",
"spatialreference": "",
"srs":
{
  "compoundwkt": "",
  "horizontal": "",
  "isgeocentric": false,
  "isgeographic": false,
  "prettycompoundwkt": "",
  "prettywkt": "",
  "proj4": "",
  "units":
  {
    "horizontal": "",
    "vertical": ""
  },
  "vertical": "",
  "wkt": ""
},
"system_id": ""
},
"filename": "1.2-with-color.las",
>tag": "readers.las1",
"type": "readers.las"
},
{
  "execution_metadata":
  {
    "area": 40981005.83,
    "boundary": "MULTIPOLYGON (((636019.34031678 847308.55730142,
→ 639990.93904966 850748.06269858, 638998.03936644 855907.32079433,
→ 633040.64126713 852467.81539716, 636019.34031678 847308.55730142)))"
}

```

```

    "density": 2.598764912e-05,
    "edge_length": 0,
    "estimated_edge": 3439.505397,
    "hex_offsets": "MULTIPOINT (0 0, -992.9 1719.75, 0 3439.51, ↴
↳ 1985.8 3439.51, 2978.7 1719.75, 1985.8 0)",
    "sample_size": 5000,
    "threshold": 10
  },
  "inputs":
  [
    "readers.las1"
  ],
  "tag": "filters.hexbin1",
  "threshold": "10",
  "type": "filters.hexbin"
},
{
  "filename": "file-output.las",
  "inputs":
  [
    "filters.hexbin1"
  ],
  "tag": "writers.las1",
  "type": "writers.las"
}
]
}

```

In addition, if you have defined a writer you will have the usual point data output file.

Example

```
{
  "pipeline": [
    "1.2-with-color.las",
    {
      "type": "filters.hexbin",
      "threshold": 10
    },
    "file-output.las"
  ]
}
```

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?
[Default: **5000**]

threshold Number of points that have to fall within a hexbin before it is considered “in” the data set. [Default: **15**]

precision Coordinate precision to use in writing out the well-known text of the boundary polygon. [Default: **8**]

7.4.22 filters.icp

The ICP filter uses the [PCL’s Iterative Closest Point \(ICP\)](#) (http://docs.pointclouds.org/trunk/classpcl_1_1_iterative_closest_point.html) 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 change 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.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Examples

```
{  
  "pipeline": [  
    "fixed.las",  
    "moving.las",  
    {  
      "type": "filters.icp"  
    },  
    "output.las"  
  ]  
}
```

To get the transform matrix, you’ll need to use the `--metadata` option:

```
$ pdal pipeline icp-pipeline.json --metadata icp-metadata.json
```

The metadata output might start something like:

```
{
  "stages": [
    {
      "filters.icp": [
        {
          "converged": true,
          "fitness": 0.01953125097,
          "transform": [
            -0.375 8.9407e-08 1 2.60209e-18 -1.97906e-09
            ↵ 98492e -10 -5.58794e-09 1 5.58794e-09 -0.5625 6.
            ↵ 0 0 1
          ]
        }
      ]
    }
  ]
}
```

See also:

filters.transformation (page 180) to apply a transform to other points.

Options

None.

7.4.23 filters.iqr

The `filters.iqr` filter automatically crops the input point cloud based on the distribution of points in the specified dimension. Specifically, we choose the method of Interquartile Range (IQR). The 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 `filters.iqr` 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.

```
{  
  "pipeline": [  
    "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.

7.4.24 filters.kdistance

The K-Distance filter creates a new attribute `KDistance` that contains the Euclidean distance to a point's k-th nearest neighbor.

Default Embedded Stage

This stage is enabled by default

Example

```
{  
  "pipeline": [  
    "input.las",  
    {  
      "type": "filters.kdistance",  
      "k": 8  
    }  
  ]  
}
```

```
        },
        {
            "type": "writers.bpf",
            "filename": "output.las",
            "output_dims": "X,Y,Z,KDistance"
        }
    ]
}
```

Options

k The number of k nearest neighbors. [Default: **10**]

7.4.25 filters.locate

The Locate filter searches the specified dimension 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.

```
{
    "pipeline": [
        "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.

7.4.26 filters.lof

Local Outlier Factor (LOF) 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 [\[Breunig2000\]](#) (page 457).

The filter creates three new dimensions, all of which are doubles. The `KDistance` dimension records the Euclidean distance between a point and its k -th nearest neighbor (the number of k neighbors is set with the `minpts` 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, where k is set with the `minpts` option.

In practice, setting the `minpts` 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` values, which users of `filters.lof` 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 uses `filters.lof` to compute 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.

```
{  
  "pipeline": [  
    ...  
  ]  
}
```

```
"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**]

7.4.27 filters.mad

The `filters.mad` filter automatically crops the input point cloud based on the distribution of points in the specified dimension. 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.

```
{  
  "pipeline": [  
    "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.

7.4.28 filters.matlab

This filter allows [Matlab](https://www.mathworks.com/products/matlab.html) (<https://www.mathworks.com/products/matlab.html>) software to be embedded in a [Pipeline](#) (page 41) that interacts with struct array of the data and allows you to modify those points. Additionally, some global [Metadata](#) (page 353) is also available that Matlab functions can interact with.

Intensity (1)

The interpreter must exit and always set `ans==true` upon success. If `ans==false` an error would be thrown and the [Pipeline](#) (page 41) exited.

See also:

[writers.matlab](#) (page 89) can be used to write .mat files.

Note: [filters.matlab](#) (page 144) 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

```
{
  "pipeline": [
    {
      "filename": "test\\data\\las\\1.2-with-color.las",
      "type": "readers.las"
    },
    {
      "type": "filters.matlab",
      "script": "matlab.m"
    },
    {
      "filename": "out.las",
      "type": "writers.las"
    }
  ]
}
```

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 [OPTIONAL, defaults PDAL]

7.4.29 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. Notably, dimensions are not initialized when points merged from various sources do not have dimensions in common.

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).

```
{  
  "pipeline": [  
    "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”.

```
{  
  "pipeline": [  
    "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”.

```
{
  "pipeline": [
    "utm1.las",
    "utm2.las",
    {
      "type" : "filters.merge"
    },
    "utm3.las",
    "out#.las"
  ]
}
```

7.4.30 filters.mongus

Filter ground returns using the approach outlined in [\[Mongus2012\]](#) (page 457).

Note: Our implementation of Mongus is in an alpha state. We'd love to have you kick the tires and provide feedback, but do not plan on using this in production.

The current implementation of `filters.mongus` differs slightly from the original paper. We weren't too happy with the criteria for how control points at the current level are compared against the TPS at the previous scale and were exploring some alternate metrics.

Some warts about the current implementation:

- It writes a bunch of intermediate/debugging outputs to the current directory while processing. This should be made optional and then eventually go away.
- We require specification of a max level, whereas the original paper automatically determined an appropriate max level.

Default Embedded Stage

This stage is enabled by default

Example

The sample pipeline below uses `filters.mongus` to segment ground and non-ground returns, writing only the ground returns to the output file.

```
{  
  "pipeline": [  
    "input.las",  
    {  
      "type": "filters.mongus",  
      "extract": true  
    },  
    "output.laz"  
  ]  
}
```

Options

cell Cell size. [Default: **1.0**]

classify Apply classification labels (i.e., ground = 2)? [Default: **true**]

extract Extract ground returns (non-ground returns are cropped)? [Default: **false**]

k Standard deviation multiplier to be used when thresholding values. [Default: **3.0**]

l Maximum level in the hierarchical decomposition. [Default: **8**]

7.4.31 filters.mortonorder

Sorts the XY data using [Morton ordering](http://en.wikipedia.org/wiki/Z-order_curve) (http://en.wikipedia.org/wiki/Z-order_curve).

Default Embedded Stage

This stage is enabled by default

Example

```
{  
  "pipeline": [  
    "uncompressed.las",  
    {  
      "type": "filters.mortonorder"  
    },  
    {  
      "type": "writers.las",  
      "filename": "compressed.laz",  
      "compression": true  
    }  
  ]  
}
```

```
    }
]
}
```

Notes

7.4.32 filters.movingleastssquares

The Moving Least Squares filter passes data through the Point Cloud Library ([PCL](http://www.pointclouds.org) (<http://www.pointclouds.org>)) MovingLeastSquares algorithm.

MovingLeastSquares is an implementation of the MLS (Moving Least Squares) algorithm for data smoothing and improved normal estimation described in [\[Alexa2003\]](#) (page 457). It also contains methods for upsampling the resulting cloud based on the parametric fit.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Example

```
{
  "pipeline": [
    "input.las",
    {
      "type": "filters.movingleastssquares"
    },
    {
      "type": "writers.las",
      "filename": "output.las"
    }
  ]
}
```

Options

None at the moment. Relying on defaults within PCL.

7.4.33 filters.normal

`filters.normal` 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 λ_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`. For more information see
https://eigen.tuxfamily.org/dox/classEigen_1_1SelfAdjointEigenSolver.html.

Normals will be automatically flipped towards the viewpoint to be consistent. By default the viewpoint is located at the midpoint of the X and Y extents, and 1000 units above the max Z value. Users can override any of the XYZ coordinates, 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`, normals will instead be inverted such that they are oriented towards the viewpoint, regardless of the `always_up` flag. To disable all normal flipping, do not provide a `viewpoint` and set `always_up=false`.

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.

```
{  
  "pipeline": [  
    "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**]

7.4.34 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). To remove the noise points altogether, users can add a *range filter* (page 170) to their pipeline, downstream from the outlier filter.

Default Embedded Stage

This stage is enabled by default

```

{
  "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 μ_i to each of the k nearest neighbors (where k is configurable and specified by `mean_k`). Then,

$$\bar{\mu} = \frac{1}{N} \sum_{i=1}^N \mu_i$$

$$\sigma = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (\mu_i - \bar{\mu})^2}$$

A global mean $\bar{\mu}$ of these mean distances is then computed along with the standard deviation σ . From this, the threshold is computed as

$$t = \mu + m\sigma$$

where m is a user-defined multiplier specified by `multiplier`.

We now iterate over the pre-computed mean distances μ_i and compare to computed threshold value. If μ_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.

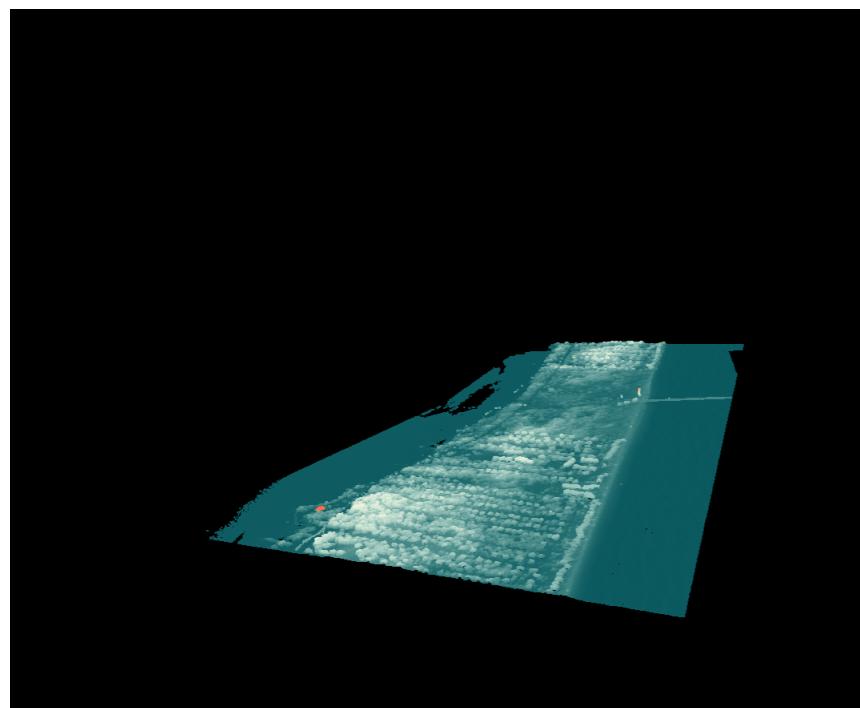
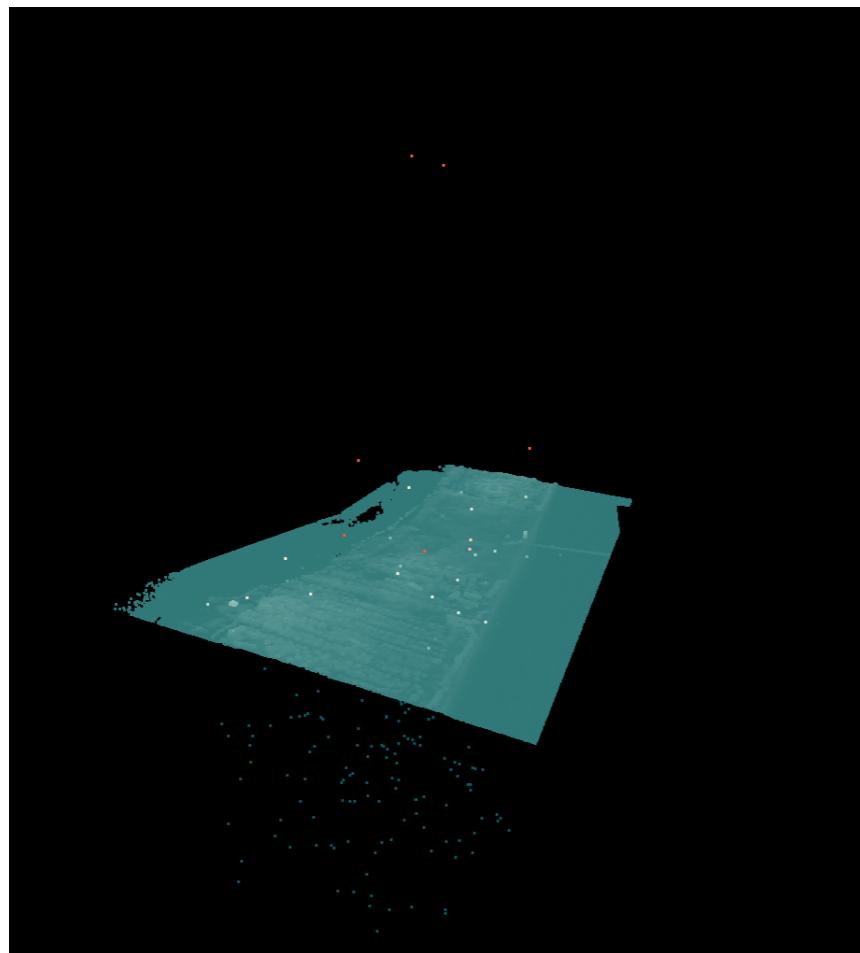
After outlier removal, the noise points are removed.

See [\[Rusu2008\]](#) (page 458) 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.

```
{
  "pipeline": [
    "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 `PointView`, this method counts the number of neighboring points k_i within radius r (specified by `radius`). If $k_i < k_{min}$, where k_{min} is the minimum number of neighbors specified by `min_k`, it is marked as an outlier.

$$\text{outlier}_i = \begin{cases} \text{true}, & \text{if } k_i < k_{min} \\ \text{false}, & \text{otherwise} \end{cases}$$

Example

The following example will mark points as outliers when there are fewer than four neighbors within a radius of 1.0.

```
{
  "pipeline": [
    "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. [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**]

7.4.35 filters.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* and a *query* 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.

```
{
  "pipeline": [
    "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) query.

```
{  
  "pipeline": [  
    "autzen-dd.las",  
    {  
      "type": "filters.overlay",  
      "dimension": "Intensity",  
      "datasource": "attributes.shp",  
      "query": "SELECT CLS FROM attributes where cls!=6",  
      "column": "CLS"  
    },  
    {  
      "filename": "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. [Defalt: first layer]

7.4.36 filterspclblock

The PCL Block filter allows users to specify a block of Point Cloud Library ([PCL](#) (<http://www.pointclouds.org>)) operations on a PDAL `PointView`, applying the necessary conversions between PDAL and PCL point cloud representations.

This filter is under active development. The current implementation serves as a proof of concept for linking PCL into PDAL and converting data. The PCL Block filter creates a PCL Pipeline object and passes it a single argument, the JSON file containing the PCL block definition. After filtering, the resulting indices can be retrieved and used to create a new PDAL `PointView` containing only those points that passed the filtering stages.

At this stage in its development, the PCL Pipeline does not allow complex operations that may change the point type (e.g., PointXYZ to PointNormal) or alter points. We will continue to look into use cases that are of value and feasible, but for now are limited primarily to PCL functions that filter or segment the point cloud, returning a list of indices of the filtered points (e.g., ground or object, noise or signal). The main reason for this design decision is that we want to avoid converting all PointView dimensions to the PCL PointCloud. In the case of an LAS reader, we may very well not want to operate on fields such as return number, but we do not want to lose this information post PCL filtering. The easy solution is to simply retain the index between the PointView and PointCloud objects and update as necessary.

See also:

See [Filtering data with PCL](#) (page 209) for more on using the PCL Block including examples.

See `pcl_json_specification` for complete details on the PCL Block JSON syntax and the filters available.

Dynamic Plugin

This stage requires a dynamic plugin to operate

Options

filename Path to external PCL JSON file describing the pipeline

methods Raw PCL JSON array describing the pipeline

PCL Block Schema

The PCL Block json object describes the filter chain to be constructed within PCL. Here is an example:

```
[  
  {  
    "name": "FilterOne",  
    "setFooParameter": "value"  
  },  
  {  
    "name": "FilterTwo",  
    "setBarParameter": false,  
    "setBounds":  
    {  
      "upper": 42,  
      "lower": 17  
    }  
  }
```

```
    }  
]
```

Implemented Filters

The list of PCL filters that are accessible through the PCL Block depends on PCL itself. PDAL is rather dumb in this respect, merely converting the PDAL `PointView` to a PCL `PointCloud` object and passing the JSON filename. The parsing of the JSON file and implementation of the PCL filters is entirely embedded within the PCL Pipeline.

A summary of the currently available filters is listed below. For full details of the filters and their parameters, see the `pcl_json_specification`.

ApproximateProgressiveMorphologicalFilter faster (and potentially less accurate) version of the **ProgressiveMorphologicalFilter**

GridMinimum assembles a local 2D grid over a given `PointCloud`, then downsamples the data

PassThrough allows the user to set min/max bounds on one dimension of the data

ProgressiveMorphologicalFilter removes nonground points to produce a bare-earth point cloud

RadiusOutlierRemoval removes outliers if the number of neighbors in a certain search radius is smaller than a given K

StatisticalOutlierRemoval uses point neighborhood statistics to filter outlier data

VoxelGrid assembles a local 3D grid over a given `PointCloud`, then downsamples and filters the data

Adding a New Filter

Adding a new PCL filter to the PCLBlock ecosystem is mostly a process of judicious copying and pasting.

1. Add the filter function declaration of the form `applyMyFilter` to `PCLPipeline.h`.
2. Add the implementation of `applyMyFilter` to `PCLPipeline.hpp`.
3. Add a one-line description of the shiny new filter to this file, `filters.pclblock.rst`.
4. Add a full description of the new filter to `pcl_spec.rst`, including example JSON, all parameters, and default settings.
5. Add a test to `PCLBlockFilterTest.cpp`. Make sure each parameter is independently verified.

7.4.37 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] (page 458).

Default Embedded Stage

This stage is enabled by default

Example

```
{
  "pipeline": [
    "input.las",
    {
      "type": "filters.pmf"
    },
    {
      "type": "writers.las",
      "filename": "output.las"
    }
  ]
}
```

Notes

- `slope` controls the height threshold at each iteration. A slope of `1.0` represents a 1:1 or 45° .
- `initial_distance` 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` 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` by the difference in window size between the current and last iteration, plus the `initial_distance`. This height threshold is constant across all cells and is maxed out at the `max_distance` 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` values will result in a lot of potentially extra iteration. This parameter can have a strongly negative impact on computation performance.
- `exponential` is used to control the rate of growth of morphological window sizes toward `max_window_size`. 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 170) to the pipeline.

```
{  
  "type": "filters.range",  
  "limits": "Classification[2:2]"  
}
```

Note: [\[Zhang2003\]](#) (page 458) 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 Optional range of values to ignore.

initial_distance Initial distance. [Default: **0.15**]

last Consider only last returns (when return information is available)? [Default: **true**]

max_distance Maximum distance. [Default: **2.5**]

max_window_size Maximum window size. [Default: **33**]

slope Slope. [Default: **1.0**]

7.4.38 filters.poisson

The poisson filter passes data Mischa Kazhdan's poisson surface reconstruction algorithm. [\[Kazhdan2006\]](#) (page 457) 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

Example

```
{  
  "pipeline": [  
    "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]



Fig. 7.9: Point cloud (800,000 points)



Fig. 7.10: Reconstruction (1.8 million vertices, 3.7 million faces)

7.4.39 filters.python

The `filters.python` filter allows Python (<http://python.org/>) software to be embedded in a *Pipeline* (page 41) that interacts with a NumPy (<http://www.numpy.org/>) array of the data and allows you to modify those points. Additionally, some global *Metadata* (page 353) 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: `filters.python` (page 164) was called `filters.programmable` and `filters.predicate` before Python 1.6. The functionality of `filters.programmable` and `filters.predicate` was rolled into `filters.python` (page 164), and the filter was renamed for clarity.

Warning: Each array contains all the *Dimensions* (page 185) of the incoming `ins` point schema. Each array in the `outs` list match 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

```
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 41) exited.
2. If you want to write a dimension that might not be available, use can use one or more `add_dimension` options.

Note: To filter points based on a Python (<http://python.org/>) function, use the `filters.python` (page 164) filter.

Modification Example

```
{
  "pipeline": [
    "file-input.las",
    {
      "type": "filters.ground"
    },
    {
      "type": "filters.python",
      "script": "multiply_z.py",
      "function": "multiply_z",
      "module": "anything"
    },
    {
      "type": "writers.las",
      "filename": "file-filtered.las"
    }
  ]
}
```

The JSON pipeline file referenced the external *multiply_z.py* Python (<http://python.org/>) script, which scales up the Z coordinate by a factor of 10.

```
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.

```
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 = keep + t

outs['Mask'] = keep
return True
```

Note: `filters.range` (page 170) 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 164) is the tool you can use for prototyping point stream processing operations.

See also:

If you want to just read a `Pipeline` (page 41) of operations into a numpy array, the PDAL Python extension might be what you want. See it at <https://pypi.python.org/pypi/PDAL>

Example Pipeline

```
{
  "pipeline": [
    "file-input.las",
    {
      "type": "filters.ground"
    },
    {
      "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 185), [Metadata](#) (page 353), and coordinate system information. Additionally, the metadata object can be set by the function to modify metadata for the in-scope [filters.python](#) (page 164) [pdal::Stage](#) (page 420).

```
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.

```
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

As of PDAL 1.5, it is possible to pass an option to [filters.python](#) (page 164) and [filters.python](#) (page 164) of JSON 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 31) with substitutions.

If we needed to be able to provide the Z scaling factor of [Example Pipeline](#) (page 166) 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.

```
{  
  "pipeline": [  
    "input.las",  
    {  
      "type": "filters.python",  
      "module": "anything",  
      "function": "filter",  
      "source": "arguments.py",  
      "pdalargs": "{\"factor\":0.3048, \"an_argument\":42, \"another\":  
      \"a string\"}"  
    },  
    "output.las"  
  ]  
}
```

With that option set, you can now fetch the pdalargs dictionary in your Python script and use it:

```
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 `redirector` module is available for scripts to output to PDAL's log stream explicitly. The module handles redirecting `sys.stderr` and `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. [Example: `functions.py`]

module The Python module that is holding the function to run. [Required]

function The function to call.

source The literal `Python` (<http://python.org/>) 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.

pdalargs A JSON dictionary of items you wish to pass into the modules globals as the `pdalargs` object.

7.4.40 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 (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` option.

Default Embedded Stage

This stage is enabled by default

Example

```
{
  "pipeline": [
    "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**]

7.4.41 filters.randomize

The randomize filter reorders the points in a point view randomly.

Default Embedded Stage

This stage is enabled by default

Example

```
{  
  "pipeline": [  
    "input.las",  
    {  
      "type": "filters.randomize"  
    },  
    {  
      "type": "writers.las",  
      "filename": "output.las"  
    }  
  ]  
}
```

7.4.42 filters.range

Contents

- *filters.range* (page 170)
 - *Pipeline Example* (page 171)
 - *Command-line Example* (page 171)
 - *Options* (page 172)
 - *Ranges* (page 172)
 - * *Example 1:* (page 172)
 - * *Example 2:* (page 172)
 - * *Example 3:* (page 172)
 - * *Example 4:* (page 173)

* *Example 5:* (page 173)

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

Pipeline 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).

```
{
  "pipeline": [
    "input.las",
    {
      "type": "filters.range",
      "limits": "Z[0:100],Classification[2:2]"
    },
    {
      "type": "writers.las",
      "filename": "filtered.las"
    }
  ]
}
```

Command-line Example

The equivalent pipeline invoked via the PDAL `translate` command would be

```
$ pdal translate -i input.las -o filtered.las -f range --filters.
  ↳range.limits="Z[0:100],Classification[2:2]"
```

Options

limits A comma-separated list of *Ranges* (page 172). 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:

```
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.

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:

```
Z[10:]
```

Selects all points with a Z value greater than or equal to 10.

Example 2:

```
Classification[2:2]
```

Selects all points with a classification of 2.

Example 3:

```
Red!(20:40]
```

Selects all points with red values less than or equal to 20 and those with values greater than 40

Example 4:

```
Blue[:255]
```

Selects all points with a blue value less than 255.

Example 5:

```
Intensity![25:25]
```

Selects all points with an intensity not equal to 25.

7.4.43 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 125) to create a new dimension and stuff them there.

Note: X, Y, and Z dimensions in PDAL are carried as doubles, with their scale information applied. Set the output scale (`scale_x`, `scale_y`, or `scale_z`) on your writer to descale the data on the way out.

Many LIDAR formats store coordinate information in 32-bit address spaces, and use scaling and offsetting to ensure that accuracy is not lost while fitting the information into a limited address space. When changing projections, the coordinate values will change, which may change the optimal scale and offset for storing the data.

Default Embedded Stage

This stage is enabled by default

Streamable Stage

This stage supports streaming operations

Example 1

This pipeline reprojecs 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.

```
{  
    "pipeline": [  
        {  
            "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.

```

1  {
2      "pipeline": [
3          "./1km_6135_632.laz",
4          {
5              "type": "filters.reprojection",
6              "in_srs": "EPSG:25832",
7              "out_srs": "+init=epsg:25832 +geoidgrids=C:/data/geoids/dvr90.
8              ↪gt;x"
9          },
10         {
11             "type": "writers.las",
12             "a_srs": "EPSG:7416",
13             "filename": "1km_6135_632_DVR90.laz"
14         }
15     ]
}

```

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 input reader does not supply SRS information]

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]

7.4.44 filters.sample

The practice of performing Poisson sampling via “Dart Throwing” was introduced in the mid-1980’s by [\[Cook1986\]](#) (page 457) and [\[Dippe1985\]](#) (page 457), and has been applied to point clouds in other software [\[Mesh2009\]](#) (page 458).

The sample filter performs Poisson sampling of the input `PointView`. 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](#) (page 184)).

See also:

[filters.decimation](#) (page 119) and [filters.voxelgrid](#) (page 184) also perform decimation.

Default Embedded Stage

This stage is enabled by default

Options

radius Minimum distance between samples. [Default: **1.0**]

7.4.45 filters.smrf

Filter ground returns using the Simple Morphological Filter (SMRF) approach outlined in [\[Pingel2013\]](#) (page 457).

Default Embedded Stage

This stage is enabled by default

Example

The sample pipeline below uses `filters.smrf` to segment ground and non-ground returns, writing only the ground returns to the output file.

```
{  
  "pipeline": [  
    "input.las",  
    {  
      "type": "filters.smrf"  
    },  
    {  
      "type": "filters.range",  
      "limits": "Classification[2:2]"  
    },  
    "output.laz"  
  ]  
}
```

Options

cell Cell size. [Default: **1.0**]

cut Cut net size (`cut=0` skips the net cutting step). [Default: **0.0**]

dir Optional output directory for debugging intermediate rasters.

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**]

7.4.46 filters.sort

The sort filter orders a point view based on the values of a dimension. The sorting can be done in increasing (ascending) or decreasing (descending) order.

Default Embedded Stage

This stage is enabled by default

Example

```
{  
  "pipeline": [  
    "unsorted.las",  
    {  
      "type": "filters.sort",  
      "dimension": "X",  
      "order": "ASC"  
    },  
    "sorted.las"  
  ]  
}
```

Options

dimension The dimension on which to sort the points.

order The order in which to sort, ASC or DESC [Default: **ASC**]

7.4.47 filters.splitter

The splitter filter breaks a point cloud into square tiles of a size that you choose. The origin of the tiles is chosen arbitrarily unless specified as an 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

```
{  
  "pipeline": [  
    "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: 0.0]

7.4.48 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.

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`.

Example

```
{
  "pipeline": [
    "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.

7.4.49 filters.tail

The TailFilter returns a specified number of points from the end of the 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

Sort and extract the 100 lowest intensity points.

```
{  
  "pipeline": [  
    {  
      "type": "filters.sort",  
      "dimension": "Intensity",  
      "order": "DESC"  
    },  
    {  
      "type": "filters.tail",  
      "count": 100  
    }  
  ]  
}
```

See also:

filters.head (page 133) is the dual to *filters.tail* (page 179).

Options

count Number of points to return. [Default: **10**]

7.4.50 filters.transformation

The transformation filter applies an arbitrary rotation+translation transformation, represented as a 4x4 matrix, to each xyz triplet.

The filter does *no* checking to ensure the matrix is a valid affine transformation — buyer beware.

Note: The transformation filter does not apply any spatial reference information — if spatial reference information is desired, it must be specified on another filter.

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.

```
{
  "pipeline": [
    "untransformed.las",
    {
      "type": "filters.transformation",
      "matrix": "0 -1 0 1 1 0 0 2 0 0 0 1 3 0 0 0 1"
    },
    {
      "type": "writers.las",
      "filename": "transformed.las"
    }
  ]
}
```

Options

matrix A whitespace-delimited transformation matrix. The matrix is assumed to be presented in row-major order. Only matrices with sixteen elements are allowed.

Notes

The transformation filter does not apply any spatial reference information — if spatial reference information is desired, it must be specified on another filter.

7.4.51 filters.voxelcenternearestneighbor

VoxelCenterNearestNeighbor 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](#) (page 184). 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

```
{  
  "pipeline": [  
    "input.las",  
    {  
      "type": "filters.voxelcenternearestneighbor",  
      "cell": 10.0  
    },  
    "output.las"  
  ]  
}
```

See also:

[filters.voxelcentroidnearestneighbor](#) (page 183) 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**]

7.4.52 filters.voxelcentroidnearestneighbor

VoxelCentroidNearestNeighbor 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 centroid of the points within that voxel is computed. This centroid is 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](#) (page 184). However, in the case of the VoxelGrid, the centroid itself 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

```
{  
  "pipeline": [  
    "input.las",  
    {  
      "type": "filters.voxelcentroidnearestneighbor",  
      "cell": 10.0  
    },  
    "output.las"  
  ]  
}
```

See also:

[filters.voxelcenternearestneighbor](#) (page 182) 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**]

7.4.53 filters.voxelgrid

The Voxel Grid filter passes data through the Point Cloud Library ([PCL](http://www.pointclouds.org) (<http://www.pointclouds.org>)) VoxelGrid algorithm.

VoxelGrid assembles a local 3D grid over a given PointCloud, and downsamples + filters the data. The VoxelGrid class creates a *3D voxel grid* (think about a voxel grid as a set of tiny 3D boxes in space) over the input point cloud data. Then, in each *voxel* (i.e., 3D box), all the points present will be approximated (i.e., *downsampled*) with their centroid. This approach is a bit slower than approximating them with the center of the voxel, but it represents the underlying surface more accurately.

Default Embedded Stage

This stage is enabled by default

Example

```
{  
  "pipeline": [  
    "untransformed.las",  
    {  
      "type": "filters.voxelgrid"  
    },  
    {  
      "type": "writers.las",  
      "filename": "transformed.las"  
    }  
  ]  
}
```

See also:

[filters.decimation](#) (page 119) does simple every-other-X -style decimation.

Options

leaf_x Leaf size in X dimension. [Default: **1.0**]

leaf_y Leaf size in Y dimension. [Default: **1.0**]

leaf_z Leaf size in Z dimension. [Default: **1.0**]

**CHAPTER
EIGHT**

DIMENSIONS

8.1 Dimensions

PDAL dimensions describe the combination of data's type, size, and meaning. The following table provides a list of known dimension names you can use in *Filters* (page 104), *Writers* (page 79), and *Readers* (page 50) descriptions.

Name	Type	Description
Alpha	uint16	Alpha
Amplitude	float	This is the ratio of the received power to the power received at the detection limit expressed in dB
Azimuth	double	Scanner azimuth
BackgroundRadiation	float	A measure of background radiation.
Blue	uint16	Blue image channel value
ClassFlags	uint8	Class Flags
Classification	uint8	ASPRS classification. 0 for no classification. See LAS specification for details.
Curvature	double	Curvature of surface at this point
Density	double	Estimate of point density
Deviation	float	A larger value for deviation indicates larger distortion.
EchoRange	double	Echo Range
EdgeOfFlightLine	int8	Indicates the end of scanline before a direction change with a value of 1 - 0 otherwise
ElevationCentroid	double	Elevation Centroid
ElevationHigh	double	Elevation High
ElevationLow	double	Elevation Low
Flag	uint8	Flag
GpsTime	double	GPS time that the point was acquired
Green	uint16	Green image channel value
HeightAboveGround	double	Height Above Ground
Infrared	uint16	Infrared

Continued on next page

Table 8.1 – continued from previous page

Name	Type	Description
Intensity	uint16	Representation of the pulse return magnitude
InternalTime	double	Scanner's internal time when the point was acquired, in seconds
IsPpsLocked	uint8	The external PPS signal was found to be synchronized at the time of the current laser shot.
LatitudeCentroid	double	Latitude Centroid
LatitudeHigh	double	Latitude High
LatitudeLow	double	Latitude Low
LongitudeCentroid	double	Longitude Centroid
LongitudeHigh	double	Longitude High
LongitudeLow	double	Longitude Low
LvisLfid	uint64	LVIS_LFID
Mark	uint8	Mark
NormalX	double	X component of a vector normal to surface at this point
NormalY	double	Y component of a vector normal to surface at this point
NormalZ	double	Z component of a vector normal to surface at this point
NumberOfReturns	uint8	Total number of returns for a given pulse.
OffsetTime	uint32	Milliseconds from first acquired point
OriginId	uint32	A file source ID from which the point originated. This ID is global to a derivative dataset which may be aggregated from multiple files.
PassiveSignal	int32	Relative passive signal
PassiveX	double	Passive X footprint
PassiveY	double	Passive Y footprint
PassiveZ	double	Passive Z footprint
Pdop	float	GPS PDOP (dilution of precision)
Pitch	float	Pitch in degrees
PointId	uint32	An explicit representation of point ordering within a file, which allows this usually-implicit information to be preserved when re-ordering points.
PointSourceId	uint16	File source ID from which the point originated. Zero indicates that the point originated in the current file
PulseWidth	float	Laser received pulse width (digitizer samples)
Red	uint16	Red image channel value
Reflectance	float	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.
ReflectedPulse	int32	Relative reflected pulse signal strength
ReturnNumber	uint8	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

Continued on next page

Table 8.1 – continued from previous page

Name	Type	Description
Roll	float	Roll in degrees
ScanAngleRank	float	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
ScanChannel	uint8	Scan Channel
ScanDirectionFlag	uint8	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
ShotNumber	uint64	Shot Number
StartPulse	int32	Relative pulse signal strength
UserData	uint8	Unspecified user data
WanderAngle	double	Wander Angle
X	double	X coordinate
XBodyAccel	double	X Body Acceleration
XBodyAngRate	double	X Body Angle Rate
XVelocity	double	X Velocity
Y	double	Y coordinate
YBodyAccel	double	Y Body Acceleration
YBodyAngRate	double	Y Body Angle Rate
YVelocity	double	Y Velocity
Z	double	Z coordinate
ZBodyAccel	double	Z Body Acceleration
ZBodyAngRate	double	Z Body Angle Rate
ZVelocity	double	Z Velocity

**CHAPTER
NINE**

PYTHON

9.1 Python

PDAL provides Python support in two significant ways. First it [embeds](https://docs.python.org/3/extending/embedding.html) (<https://docs.python.org/3/extending/embedding.html>) Python to allow you to write Python programs that interact with data using [*filters.python*](#) (page 164) filter. Second, it [extends](https://docs.python.org/3/extending/extending.html) (<https://docs.python.org/3/extending/extending.html>) 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/) (<http://www.numpy.org/>) support. PDAL's data is provided to both the filters and the extension as Numpy arrays.

9.1.1 Versions

PDAL supports both Python 2.7 and Python 3.4+. integration tests Python 2.7 on both Linux and Windows. Python 3 is used by a number of developers for adhoc development and testing.

9.1.2 Embed

PDAL allows users to embed Python functions inline with other [*Pipeline*](#) (page 41) 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.

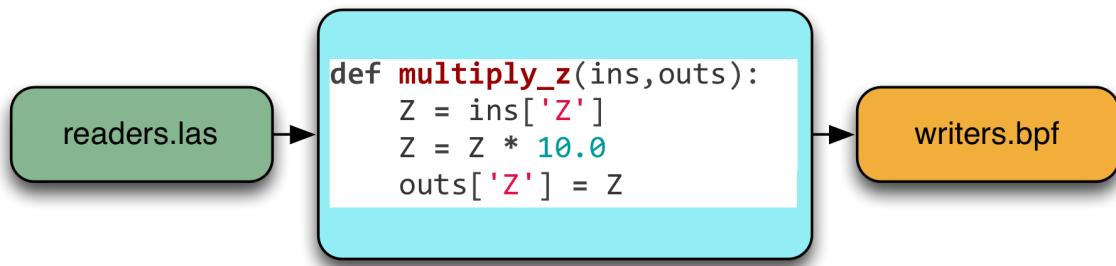


Fig. 9.1: Embedding a Python function to take Z values read from a *readers.las* (page 59) and then output them to a *writers.bpf* (page 79).

9.1.3 Extend

PDAL provides a Python extension that gives users access to executing *Pipeline* (page 41) 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 JSON *Pipeline* (page 41) 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:

```

json = """
{
  "pipeline": [
    "1.2-with-color.las",
    {
      "type": "filters.sort",
      "dimension": "X"
    }
  ]
}"""

import pdal
pipeline = pdal.Pipeline(json)
pipeline.validate() # check if our JSON and options were good
pipeline.loglevel = 8 #really noisy
count = pipeline.execute()
arrays = pipeline.arrays
metadata = pipeline.metadata
log = pipeline.log
  
```

Installation

PDAL Python bindings require a working PDAL install ([PDAL](#) (page 334)) and then installation of the Python extension. The extension lives on PyPI (<https://pypi.python.org/pypi/PDAL>) at <https://pypi.python.org/pypi/PDAL> and you should use that version as your canonical Python extension install.

Install from local

In the source code of PDAL there is a `python` folder, you have to enter there and run

```
python setup.py build  
# this should be run as administrator/super user  
python setup.py install
```

Install from repository

The second method to install the PDAL Python extension is to use `pip` (<https://pip.pypa.io/en/stable/>) or `easy_install` (<https://pypi.python.org/pypi/setuptools>), you have to run the command as administrator.

```
pip install PDAL
```

Note: To install pip please read [here](https://pip.pypa.io/en/stable/installing/) (<https://pip.pypa.io/en/stable/installing/>)

CHAPTER
TEN

TUTORIALS

10.1 Tutorials

This section provides a collection of tutorials on how to use the PDAL *Applications* (page 23) and *Pipelines* (page 41) to process data.

Note: Users looking for documentation on how to contribute to PDAL should look [here](#) (page 323) and users looking to use the PDAL API in their own applications should look [here](#) (page 386).

10.1.1 Reading with PDAL

Author Bradley Chambers

Contact brad.chambers@gmail.com

Date 01/21/2015

Contents

- *Reading with PDAL* (page 193)
 - *Introduction* (page 194)
 - *A basic inquiry example* (page 194)
 - *A conversion example* (page 195)
 - * *Metadata* ([page 353](#)) (page 196)
 - *A pipeline (page 31) example* (page 196)
 - * *Simple conversion* (page 197)

* *Loop a directory and filter it through a pipeline* (page 197)

This tutorial will be presented in two parts – the first being an introduction to the command-line utilities that can be used to perform processing operations with PDAL, and the second being an introductory C++ tutorial of how to use the [PDAL API](#) (page 387) to accomplish similar tasks.

Introduction

PDAL is both a C++ library and a collection of command-line utilities for data processing operations. While it is similar to [LAStools](#) (<http://lastools.org>) in a few aspects, and borrows some of its lineage in others, the PDAL library is an attempt to construct a library that is primarily intended as a data translation library first, and a exploitation and filtering library second. PDAL exists to provide an abstract API for software developers wishing to navigate the multitude of point cloud formats that are out there. Its value and niche is explicitly modeled after the hugely successful [GDAL](#) (<http://www.gdal.org>) library, which provides an abstract API for data formats in the GIS raster data space.

A basic inquiry example

Our first example to demonstrate PDAL's utility will be to simply query an [ASPRS LAS](#) (http://www.asprs.org/a/society/committees/standards/lidar_exchange_format.html) file to determine the data that are in it in the very first point.

Note: The `interesting.las`

(<https://github.com/PDAL/PDAL/blob/master/test/data/las/interesting.las?raw=true>) file in these examples can be found on github.

`pdal info` outputs JavaScript [JSON](#) (<http://www.json.org/>).

```
$ 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 one file format to another can be a hairy topic. You should expect *leakage* of details of data in the source format as it is converted to the destination format. [Metadata](#) (page 353), file organization, and data themselves may not be able to be represented as you move from one format to another. Conversion is by definition lossy, if not in terms of the actual data themselves, but possibly in terms of the auxiliary data the format also carries.

It is also important to recognize that both fixed and flexible point cloud formats exist, and conversion of flexible formats to fixed formats will often leak. The dimensions might even match in terms of type or name, but not in terms of width or interpretation.

See also:

See [pdal::Dimension](#) (page 395) for details on PDAL dimensions.

```
$ pdal translate interesting.las output.txt
```

```

"X", "Y", "Z", "Intensity", "ReturnNumber", "NumberOfReturns",
→ "ScanDirectionFlag", "EdgeOfFlightLine", "Classification",
→ "ScanAngleRank", "UserData", "PointSourceId", "Time", "Red", "Green",
→ "Blue"
637012.24, 849028.31, 431.66, 143, 1, 1, 1, 0, 1, -9, 132, 7326, 245381, 68, 77, 88
636896.33, 849087.70, 446.39, 18, 1, 2, 1, 0, 1, -11, 128, 7326, 245381, 54, 66, 68
636784.74, 849106.66, 426.71, 118, 1, 1, 0, 0, 1, -10, 122, 7326, 245382, 112, 97,
→ 114
636699.38, 848991.01, 425.39, 100, 1, 1, 0, 0, 1, -6, 124, 7326, 245383, 178, 138,
→ 162
636601.87, 849018.60, 425.10, 124, 1, 1, 1, 0, 1, -4, 126, 7326, 245383, 134, 104,
→ 134

```

```
636451.97,849250.59,435.17,48,1,1,0,0,1,-9,122,7326,245384,99,85,95  
...
```

The text format, of course, is the ultimate flexible-definition format – at least for the point data themselves. For the other header information, like the spatial reference system, or the [ASPRS LAS](http://www.asprs.org/a/society/committees/standards/lidar_exchange_format.html) (http://www.asprs.org/a/society/committees/standards/lidar_exchange_format.html) [UUID](http://en.wikipedia.org/wiki/Universally_unique_identifier) (http://en.wikipedia.org/wiki/Universally_unique_identifier), the conversion leaks. In short, you may need to preserve some more information as part of your conversion to make it useful down the road.

Metadata

PDAL transmits this other information in the form of [Metadata](#) (page 353) that is carried per-stage throughout the PDAL [processing pipeline](#) (page 41). We can capture this metadata using the [info](#) (page 27) utility.

```
$ pdal info --metadata interesting.las
```

This produces metadata that looks like this. You can use your [JSON](#) (<http://www.json.org/>) 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.

See also:

[Metadata](#) (page 353) contains much more detail of metadata workflow in PDAL.

A pipeline example

The full power of PDAL comes in the form of [pipeline](#) (page 31) invocations. While [translate](#) (page 36) provides some utility as far as simple conversion of one format to another, it does not provide much power to a user to be able to filter or alter data as they are converted. Pipelines are the way 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](#) (page 41), but the [Pipeline](#) (page 41) document contains more detailed exposition of the topic.

Note: The [pipeline](#) (page 31) document contains detailed examples and background information.

The [pipeline](#) (page 31) PDAL utility is one that takes in a `.json` file containing [pipeline](#) (page 31) description that defines a PDAL processing pipeline. Options can be given at each [pdal::Stage](#) (page 420) of the pipeline to affect different aspects of the processing pipeline, and stages may be chained together into multiple combinations to have varying effects.

Simple conversion

The following [JSON](http://www.json.org/) (<http://www.json.org/>) document defines a *Pipeline* (page 41) that takes the `file.las` [ASPRS LAS](http://www.asprs.org/a/society/committees/standards/lidar_exchange_format.html) (http://www.asprs.org/a/society/committees/standards/lidar_exchange_format.html) file and converts it to a new file called `output.las`.

```
{  
  "pipeline": [  
    "file.las",  
    "output.las"  
  ]  
}
```

Loop a directory and filter it through a pipeline

This little 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 -P20 -I{} pdal pipeline -i /path/to/  
↳ proj.json --readers.las.filename={} .las --writers.las.  
↳ filename=output/{}.laz
```

10.1.2 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 197)
 - *Introduction* (page 198)
 - *LAS Versions* (page 198)
 - *Spatial Reference System* (page 199)
 - *Point Formats* (page 202)
 - *Extra Dimensions* (page 203)
 - *Required Header Fields* (page 204)

- *Coordinate Scaling* (page 205)
- *Compression* (page 206)
- *PDAL Metadata* (page 208)

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 59) and *writers.las* (page 85) 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 85) 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.

```
1  {
2      "pipeline": [
3          {
4              "type" : "readers.las",
5              "filename" : "input.las"
6          },
7          {
```

```
8     "type" : "writers.las",
9     "minor_version": 1,
10    "filename" : "output.las"
11  }
12 ]
13 }
```

Note: PDAL defaults to writing a LAS 1.2 version if no `minor_version` is specified or the `forward` option of [writers.las](#) (page 85) is not used to carry along a version from a previously read file.

Spatial Reference System

LAS 1.0 to 1.3 used [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 more expressive and 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 85) 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 85) to EPSG:4326.

```
1 {
2   "pipeline": [
3     {
4       "type" : "readers.las",
5       "filename" : "input.las"
6     },
7     {
8       "type" : "writers.las",
9       "a_srs": "EPSG:4326",
10      "filename" : "output.las"
11    }
12  ]
13 }
```

Note: Remember to set `offset_x`, `offset_y`, `scale_x`, and `scale_y` values to something appropriate if your 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 206) with *LASzip* (<http://laszip.org>).

Vertical Datum Example

Vertical coordinate control is important in [LiDAR](https://en.wikipedia.org/wiki/Lidar) (<https://en.wikipedia.org/wiki/Lidar>) and PDAL supports assignment and reprojection/transform of vertical coordinates using [Proj.4](http://proj4.org) (<http://proj4.org>) and [GDAL](http://gdal.org/) (<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/gravitymod/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:

```
$ gdalsrsinfo "EPSG:4326+3855"
```

```
COMPD_CS["WGS 84 + EGM2008 geoid height",
  GEOGCS["WGS 84",
    DATUM["WGS_1984",
      SPHEROID["WGS 84", 6378137, 298.257223563,
        AUTHORITY["EPSG", "7030"]],
      AUTHORITY["EPSG", "6326"]],
    PRIMEM["Greenwich", 0,
      AUTHORITY["EPSG", "8901"]],
    UNIT["degree", 0.0174532925199433,
      AUTHORITY["EPSG", "9122"]],
    AUTHORITY["EPSG", "4326"]],
  VERT_CS["EGM2008 geoid height",
    VERT_DATUM["EGM2008 geoid", 2005,
      AUTHORITY["EPSG", "1027"]],
    EXTENSION["PROJ4_GRID", "egm08_25.gtx"]],
  UNIT["metre", 1,
    AUTHORITY["EPSG", "9001"]],
  AXIS["Up", UP],
  AUTHORITY["EPSG", "3855"]]
```

As in [Assignment Example](#) (page 199), 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.

```

1  {
2      "pipeline": [
3          {
4              "type" : "readers.las",
5              "filename" : "input.las"
6          },
7          {
8              "type" : "writers.las",
9              "a_srs": "EPSG:26915+5703",
10             "filename" : "output.las"
11         }
12     ]
13 }
```

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 to escape or massage any input data to make it be valid JSON, however.

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 173).

```

1  {
2      "pipeline": [
3          {
4              "type" : "readers.las",
5              "filename" : "input.las"
6          },
7          {
8              "type": "filters.reprojection",
9              "out_srs": "EPSG:26915"
10         },
11         {
12             "type" : "writers.las",
13         }
14     ]
15 }
```

```
13         "filename" : "output.las"
14     }
15 ]
16 }
```

Warning: When you are transforming coordinates, you might need to set the `scale_x`, `scale_y`, `offset_x`, and `offset_y` values to something reasonable for your output coordinate system.

Note: If the input data doesn't specify a projection, you must specify the `in_srs` option of [`filters.reprojection`](#) (page 173). `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 185) that a LAS file stores for each point in the file. For any given 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 185) stored for each point in a LAS file. All point formats specify the following dimensions as part of a point record:

Table 10.1: Base LAS Dimensions

X	Y	Z
Intensity	ReturnNumber	NumberOfReturns
ScanDirectionFlag	EdgeOfFlightLine	Classification
ScanAngleRank	UserData	PointSourceId

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.

```

1  {
2      "pipeline": [
3          {
4              "type" : "readers.las",
5              "filename" : "input.las"
6          },
7          {
8              "type" : "writers.las",
9              "forward": "all",
10             "dataformat_id": 0,
11             "filename" : "output.las"
12         }
13     ]
14 }
```

Note: The [LASzip](http://laszip.org) (<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 sent of *Dimensions* (page 185) a file must store, but softwares 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

The following example will write a LAS 1.4 file with all dimensions written into the file along with a description of those dimensions in the Extra Bytes VLR in the 1.4 file:

```

1  {
2      "pipeline": [
3          "some_non_las_file",
4          {
5              "type" : "writers.las",
6              "extra_dims": "all",
7              "minor_version" : "4",
8              "filename" : "output.las"
9          }
10 }
```

```
10     ]
11 }
```

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 85) 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 159) to reassign them. During this scenario, we'd like to keep all of the other LAS header information, such as *Variable Length Records* (page 207), extent information, and format settings.

```
1 {
2     "pipeline": [
3         {
4             "type" : "readers.las",
5             "filename" : "input.las"
6         },
7         {
8             "type" : "filters.assign",
9             "assignment" : "Classification[0:32]=0"
10        },
11        {
12            "type" : "filters.pmf",
13            "cell_size" : 2.5,
14            "approximate" : false,
15            "max_distance" : 25
16        },
17    ]
```

```

17     {
18         "type" : "writers.las",
19         "forward": "all",
20         "filename" : "output.las"
21     }
22 ]
23 }
```

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) (<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) (<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) (<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.

```

1 {
2     "pipeline": [
3         {
4             "type" : "readers.las",
```

```
5         "filename" : "input.las"
6     } ,
7     {
8         "type" : "writers.las",
9         "scale_x": "0.0000001",
10        "scale_y": "0.0000001",
11        "scale_z": "0.01",
12        "offset_x": "auto",
13        "offset_y": "auto",
14        "offset_z": "auto",
15        "filename" : "output.las"
16    }
17 ]
18 }
```

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 59) and *writers.las* (page 85). It can be enabled by setting the `compression` option to `laszip`. LAZ support is not fully compatible with LAS 1.4 as of March 2017. A revision with 1.4 support is expected.

Compression Example

Compressing LAS data with LASzip (<http://laszip.org>) is a simple option on the *writers.las* (page 85):

```
1 {
2     "pipeline": [
3         {
4             "type" : "readers.las",
5             "filename" : "input.las"
6         },
7         {
8             "type" : "writers.las",
9             "compression": "laszip",
10            "filename" : "output.laz"
11        }
12    ]
13 }
```

Note: If the filename ends in the extension .laz but no compression option is given, the [writers.las](#) (page 85) will set the compression to laszip for you and write out a [LASzip](#) (<http://laszip.org>)-compressed file.

Variable Length Records

Variable Length Records, or VLRs, are blobs of 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.

Variable Length Records (VLRs) are how applications can insert their own data into LAS files. 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. For EVLRs, stored at the end of the file in LAS 1.4, this limit was increased to 4gb.

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 85) vlr 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.

```
1  {
2      "pipeline": [
3          "input.las",
```

```
4  {
5      "type": "writers.las",
6      "filename": "output.las",
7      "vlrs": [    {
8          "description": "A description under 32 bytes",
9          "record_id": 42,
10         "user_id": "hobu",
11         "data": "dGhpcyBpcyBzb21lIHRleHQ="
12     },
13     {
14         "description": "A description under 32 bytes",
15         "record_id": 43,
16         "user_id": "hobu",
17         "data": "dGhpcyBpcyBzb21lIG1vcmUgdGV4dA=="
18     }
19   ]
20 }
21 ]
22 }
```

PDAL Metadata

The [writers.las](#) (page 85) driver supports an option, `pdal_metadata`, that writes two *PDAL* VLRs to LAS files. The first is the equivalent of [info](#) (page 27)'s `--metadata` output. The second is a dump 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](#) (page 41) used to construct the file and all of its [Metadata](#) (page 353) 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/misic/las-key-list.html>).

```
1  {
2      "pipeline": [
3          {
4              "type": "readers.las",
5              "filename": "input.las"
6          },
7          {
8              "type": "writers.las",
9              "filename": "output.las"
10             "vlrs": [
11                 {
12                     "description": "A description under 32 bytes",
13                     "record_id": 42,
14                     "user_id": "hobu",
15                     "data": "dGhpcyBpcyBzb21lIHRleHQ="
16                 },
17                 {
18                     "description": "A description under 32 bytes",
19                     "record_id": 43,
20                     "user_id": "hobu",
21                     "data": "dGhpcyBpcyBzb21lIG1vcmUgdGV4dA=="
22                 }
23               ]
24           }
25         ]
26     }
```

```

8      "type" : "writers.las",
9      "pdal_metadata": "true",
10     "filename" : "output.laz"
11   }
12 ]
13 }
```

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.

10.1.3 Filtering data with PCL

Introduction

PDAL is both a C++ library and a collection of command-line utilities for data processing operations. While the PDAL library addresses point cloud exploitation and filtering, this takes a back seat to its primary objective of being a data translation library, helping developers to navigate the a wide variety of point cloud formats. [PCL](http://www.pointclouds.org) (<http://www.pointclouds.org>) is another C++ library that is focused on developing a rich set of point cloud processing routines, with less of a focus on formats and data translation. Acknowledging this, the PCL Block filter was developed to serve as a bridge between the two libraries, enabling rapid development of point cloud processing pipelines.

See also:

See [filters.pclblock](#) (page 156) for details on PDAL’s PCL Block filter.

Contents

- [Filtering data with PCL](#) (page 209)
 - [Introduction](#) (page 209)
 - [Quick Start](#) (page 210)
 - [PDAL Pipeline kernel](#) (page 210)
 - [PDAL PCL kernel](#) (page 211)
 - [Examples](#) (page 212)
 - * [Simple point cloud cropping](#) (page 212)
 - * [Point cloud cropping with outlier removal](#) (page 213)

* *Ground return filtering* (page 213)

Quick Start

The *Quickstart* (page 15) document describes how to use PDAL with Docker, which includes built-in PCL support. After you have worked through that document, you should be able to run any PDAL PCL operations.

PDAL Pipeline kernel

Note: A full description of the PDAL pipeline concept is beyond the scope of this tutorial but the *Pipeline* (page 41), *pipeline* (page 31), and *Reading with PDAL* (page 193) documents contain detailed examples and background information.

The *filters.pclblock* (page 156) is implemented as a PDAL filter stage and as such is easily accessed via the PDAL pipeline. It accepts a single, required option - the name of the **JSON** (<http://www.json.org/>) file describing the PCL Block.

A sample pipeline JSON which reads/writes LAS and has a single PCL Block filter is shown below.

```
{  
  "pipeline": [  
    "autzen-point-format-3.las",  
    {  
      "type": "filters.pclblock",  
      "filename": "passthrough.json"  
    },  
    "foo.las"  
  ]  
}
```

And is run from the command line thusly.

```
$ pdal pipeline passthrough.json
```

This simple pipeline reads the input LAS (`autzen-point-format-3.las`), passes it through the PCL Block (`passthrough.json`), and writes the output LAS (`foo.las`).

When run, it should produce output similar to this:

```
Requested to read 106 points  
Requested to write 106 points
```

```
0
Processing /home/vagrant/pdal/test/data/filters/pcl/passthrough.json

-----
→
NAME: PassThroughExample ()
HELP:
AUTHOR:
-----
→
106 points copied

Step 1) PassThrough

Field name: z
Limits: 410.000000, 440.000000

76(writers.las DEBUG: 3): Wrote 81 points to the LAS file
.100
```

PDAL PCL kernel

For users that would like to bypass the creation (and subsequent modification) of the pipeline JSON for every file they wish to process, there is another option: the `pdal pcl` command.

```
$ pdal pcl -i /path/to/input/las -p /path/to/pcl/block/json -o /path/
→/to/output/las
```

This is functionally equivalent to the original *pdal pipeline* command, but does not afford the flexibility of constructing the pipeline (i.e., none the other PDAL filters are accessible).

The same can be accomplished with the `pdal pcl` command. The basic syntax for the command is

```
$ pdal pcl -i <input cloud> -p <PCL Block JSON> -o <output cloud>
```

where the JSON file specified with `-p` is the same file that would be embedded in the pipeline JSON file. This can be useful when the pipeline does not change frequently, but the input/output filenames do.

For example, the above *pdal pipeline* example can be written with *pdal pcl* like this:

```
$ cd pdal # your PDAL source tree
$ cd test/data
$ ../../bin/pdal pcl -i autzen/autzen-point-format-3.las -p filters/
→pcl/example_PassThrough_1.json -o ../temp/foo.las -v4
```

This should produce the output

```
Requested to read 106 points
Requested to write 106 points
0
Processing /home/vagrant/pdal/test/data/filters/pcl/passthrough.json

-----
→
NAME: PassThroughExample ()
HELP:
AUTHOR:

-----
→
106 points copied

Step 1) PassThrough

    Field name: z
    Limits: 410.000000, 440.000000

76(writers.las DEBUG: 3): Wrote 81 points to the LAS file
.100
```

Examples

Simple point cloud cropping

The power of the PCL Block is really exposed through the JSON description. In this example, we apply a single PCL filter to the PointView. The [PassThrough](http://pointclouds.org/documentation/tutorials/passthrough.php) (<http://pointclouds.org/documentation/tutorials/passthrough.php>) filter removes points that lie outside a given range for the specified dimension. Here, we are asking PCL to crop the input point cloud, returning only those points with z values in the range 100 to 200.

```
[
  {
    "name": "PassThrough",
    "setFilterFieldName": "z",
    "setFilterLimits": [
      {
        "min": 410.0,
        "max": 440.0
      }
    ]
}
```

(This example is taken from the unit test *PCLBlockFilterTest_example_PassThrough_1*.)

Point cloud cropping with outlier removal

Building on the previous example, we can string together multiple PCL filtering stages, such as the [StatisticalOutlierRemoval](#)

(http://pointclouds.org/documentation/tutorials/statistical_outlier.php) filter. Note that the name field identifies the PCL filter by its class name, and furthermore that as of now only a handful of the PCL filtering options are accessible through the PCL Block. Similarly, select parameters of these classes can be set by specifying their public member functions by name.

```
[  
  {  
    "name": "PassThrough",  
    "help": "filter z values to the range [410, 440]",  
    "setFilterFieldName": "z",  
    "setFilterLimits":  
    {  
      "min": 410.0,  
      "max": 440.0  
    }  
  },  
  {  
    "name": "StatisticalOutlierRemoval",  
    "help": "apply outlier removal",  
    "setMeanK": 8,  
    "setStddevMulThresh": 0.2  
  }  
]
```

(This example is taken from the unit test *PCLBlockFilterTest_example_PassThrough_2*.)

Ground return filtering

The Progressive Morphological Filter (PMF) is an openly published approach to identifying ground vs. non-ground returns in point cloud data. An implementation of PMF is included with PCL and accessible through the PDAL's PCL Block filter.

A complete description of the algorithm can be found in the article “[A Progressive Morphological Filter for Removing Nonground Measurements from Airborne LIDAR Data](#)” (<http://users.cis.fiu.edu/~chens/PDF/TGRS.pdf>) by K. Zhang, S. Chen, D. Whitman, M. Shyu, J. Yan, and C. Zhang.

To run the PMF with default settings, the PCL Block JSON is simply:

```
[  
  {  
    "name": "ProgressiveMorphologicalFilter"  
    "setMaxWindowSize": 200,  
  }  
]
```

Additional parameters can be set by advanced users:

```
[  
  {  
    "name": "ProgressiveMorphologicalFilter",  
    "setCellSize": 1.0,  
    "setMaxWindowSize": 200,  
    "setSlope": 1.0,  
    "setInitialDistance": 0.5,  
    "setMaxDistance": 3.0,  
    "setExponential": true  
  }  
]
```

(These examples are taken from the unit tests *PCLBlockFilterTest_example_PMF_1* and *PCLBlockFilterTest_example_PMF_2*.)

See [here](#) (page 214) for a more detailed explanation of the PMF parameters.

10.1.4 Identifying ground returns using ProgressiveMorphological-Filter segmentation

Author Bradley Chambers

Contact brad.chambers@gmail.com

Date 10/28/2015

Implements the Progressive Morphological Filter for segmentation of ground points.

Note: `filters.ground` required PCL and has since been replaced by [`filters.pmf`](#) (page 159), which is a native PDAL filter. [`ground`](#) (page 26) has been retained, but now calls [`filters.pmf`](#) (page 159) under the hood as opposed to `filters.ground` and is installed as a native PDAL kernel independent of the PCL plugin. As such, the outputs shown in this tutorial may vary slightly, but the underlying algorithm is identical.

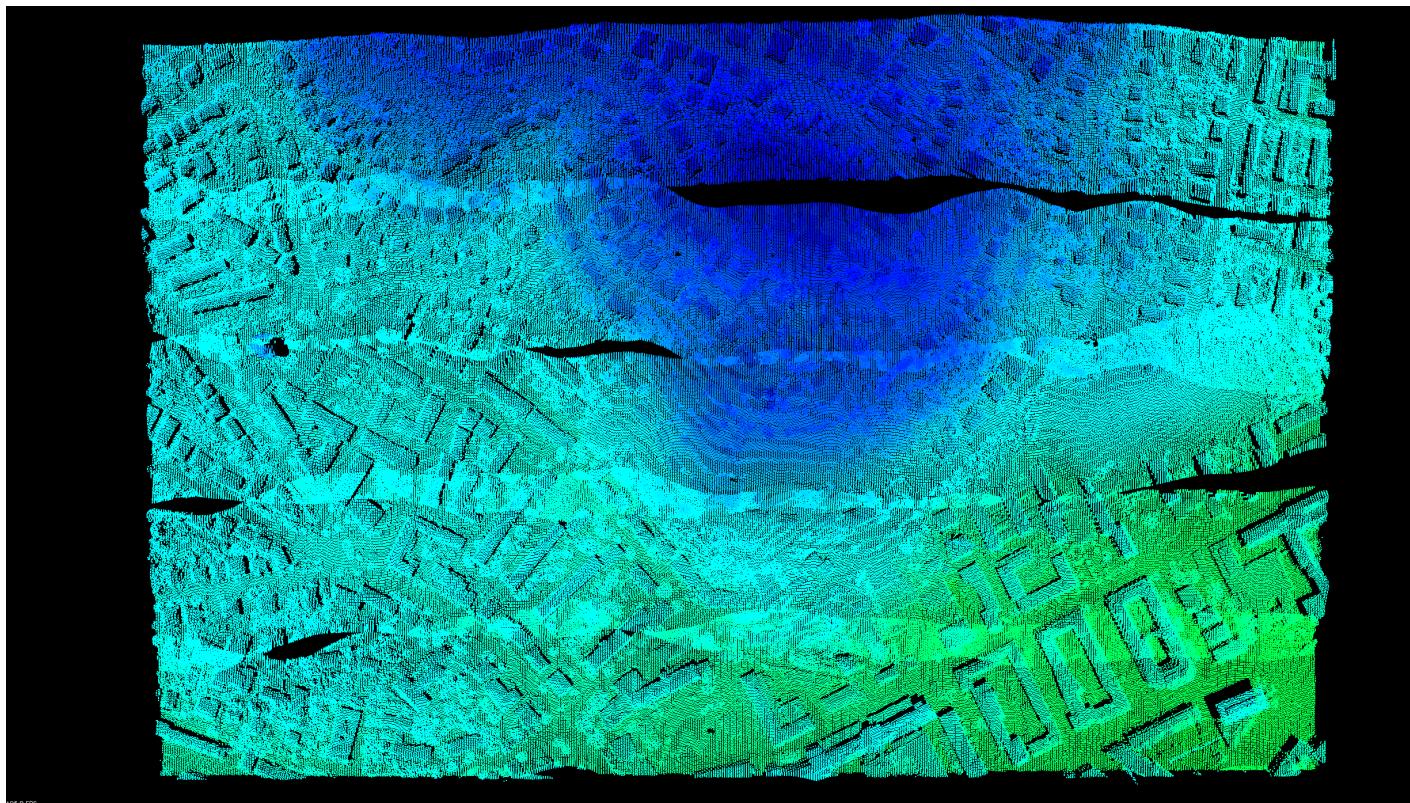
Background

A complete description of the algorithm can be found in the article “[A Progressive Morphological Filter for Removing Nonground Measurements from Airborne LIDAR Data](http://users.cis.fiu.edu/~chens/PDF/TGRS.pdf)” (<http://users.cis.fiu.edu/~chens/PDF/TGRS.pdf>) by K. Zhang, S. Chen, D. Whitman, M. Shyu, J. Yan, and C. Zhang.

For more information on how to invoke this PCL-based filter programmatically, see the [ProgressiveMorphologicalFilter](http://pointclouds.org/documentation/tutorials/progressive_morphological_filtering.php) (http://pointclouds.org/documentation/tutorials/progressive_morphological_filtering.php) tutorial on the PCL website.

We have chosen to demonstrate the algorithm using data from the 2003 report “[ISPRS Comparison of Filters](http://www.itc.nl/isprswgIII-3/filtertest/).“ For more on the data and the study itself, please see <http://www.itc.nl/isprswgIII-3/filtertest/> as well as “[Experimental comparison of filter algorithms for bare-earth extraction from airborne laser scanning point clouds](http://dx.doi.org/10.1016/j.isprsjprs.2004.05.004)” (<http://dx.doi.org/10.1016/j.isprsjprs.2004.05.004>) by G. Sithole and G. Vosselman.

First, download the dataset [CSite1_orig-utm.laz](https://raw.github.com/PDAL/data/master/isprs/CSite1_orig-utm.laz) (https://raw.github.com/PDAL/data/master/isprs/CSite1_orig-utm.laz) and save it somewhere to disk.



Using the Ground kernel

The *pdal ground* (page 26) kernel can be used to filter ground returns, allowing the user to tweak filtering parameters at the command-line.

Let's start by running `pdal ground` with the default parameters.

```
$ pdal ground -i CSitel_orig-utm.laz -o CSitel_orig-utm-ground.laz
```

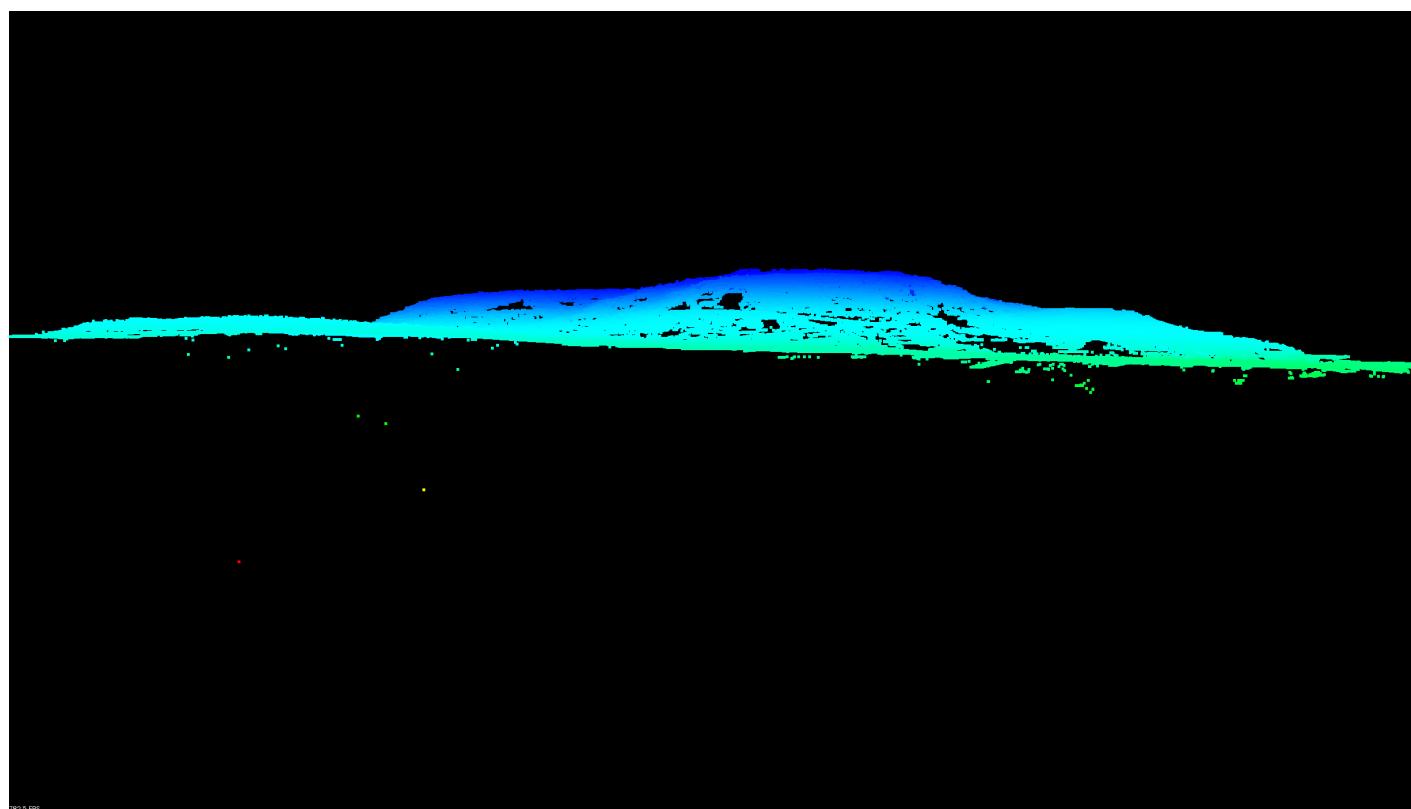
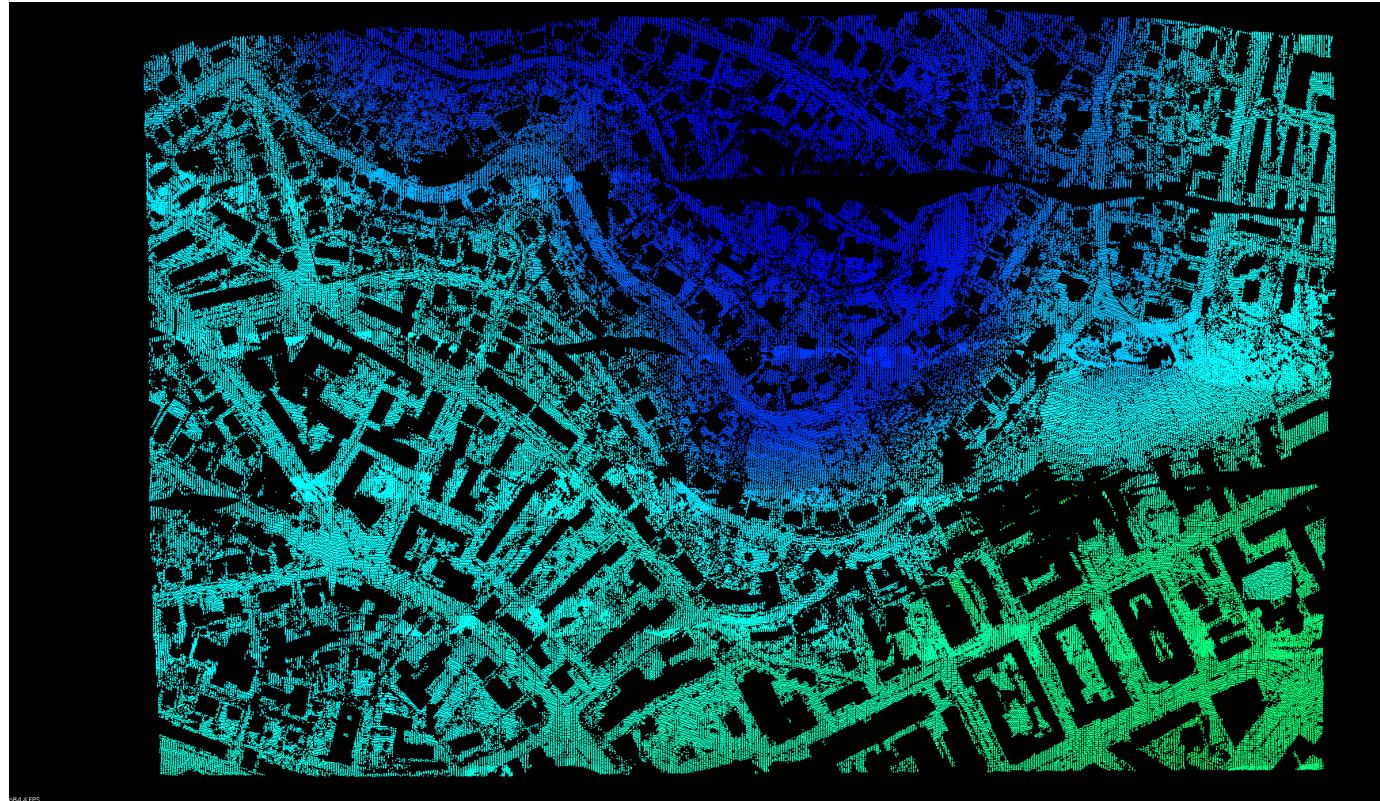
To get an idea of what's happening during each iteration, you can optionally increase the verbosity of the output. We'll try `-v4`. Here we see a summary of the parameters, along with height threshold, window size, and number of remaining ground points.

```
$ pdal ground -i CSitel_orig-utm.laz -o CSitel_orig-utm-ground.laz -v4
-----
-----
NAME:      ()
HELP:
AUTHOR:
-----
-----
process tile 0 through the pipeline

Step 1) ProgressiveMorphologicalFilter

    max window size: 33
    slope: 1.000000
    max distance: 2.500000
    initial distance: 0.150000
    cell size: 1.000000
    base: 2.000000
    exponential: true
    negative: false
    Iteration 0 (height threshold = 0.150000, window size = 3.
    ↪000000)...ground now has 872413 points
    Iteration 1 (height threshold = 2.150000, window size = 5.
    ↪000000)...ground now has 833883 points
    Iteration 2 (height threshold = 2.500000, window size = 9.
    ↪000000)...ground now has 757030 points
    Iteration 3 (height threshold = 2.500000, window size = 17.
    ↪000000)...ground now has 625333 points
    Iteration 4 (height threshold = 2.500000, window size = 33.
    ↪000000)...ground now has 580852 points
    1366408 points filtered to 580852 following progressive_
    ↪morphological filter
```

The resulting filtered cloud can be seen in this top-down and front view. When viewed from the side, it is apparent that there are a number of low noise points that have fooled the PMF filter.



To address, we introduce an alternate way to call PMF, as part of a PCL pipeline, where we preprocess with an outlier removal step. The command is nearly identical, replacing ground with pcl and adding a pipeline JSON specified with -p.

```
{  
    "pipeline": {  
        "name": "Progressive Morphological Filter with Outlier Removal",  
        "version": 1.0,  
        "filters": [{  
            "name": "StatisticalOutlierRemoval",  
            "setMeanK": 8,  
            "setStddevMulThresh": 3.0  
        }, {  
            "name": "ProgressiveMorphologicalFilter"  
        }]  
    }  
}
```

```
$ pdal pcl -i CSitel_orig-utm.laz -o CSitel_orig-utm-ground.laz -p  
→sor-pmf.json -v4
```

```
-----  
→-----  
NAME: Progressive Morphological Filter with Outlier Removal (1.0)
```

```
HELP:
```

```
AUTHOR:
```

```
-----  
→-----  
process tile 0 through the pipeline
```

```
Step 1) StatisticalOutlierRemoval
```

```
8 neighbors and 3.000000 multiplier  
1366408 points filtered to 1356744 following outlier removal
```

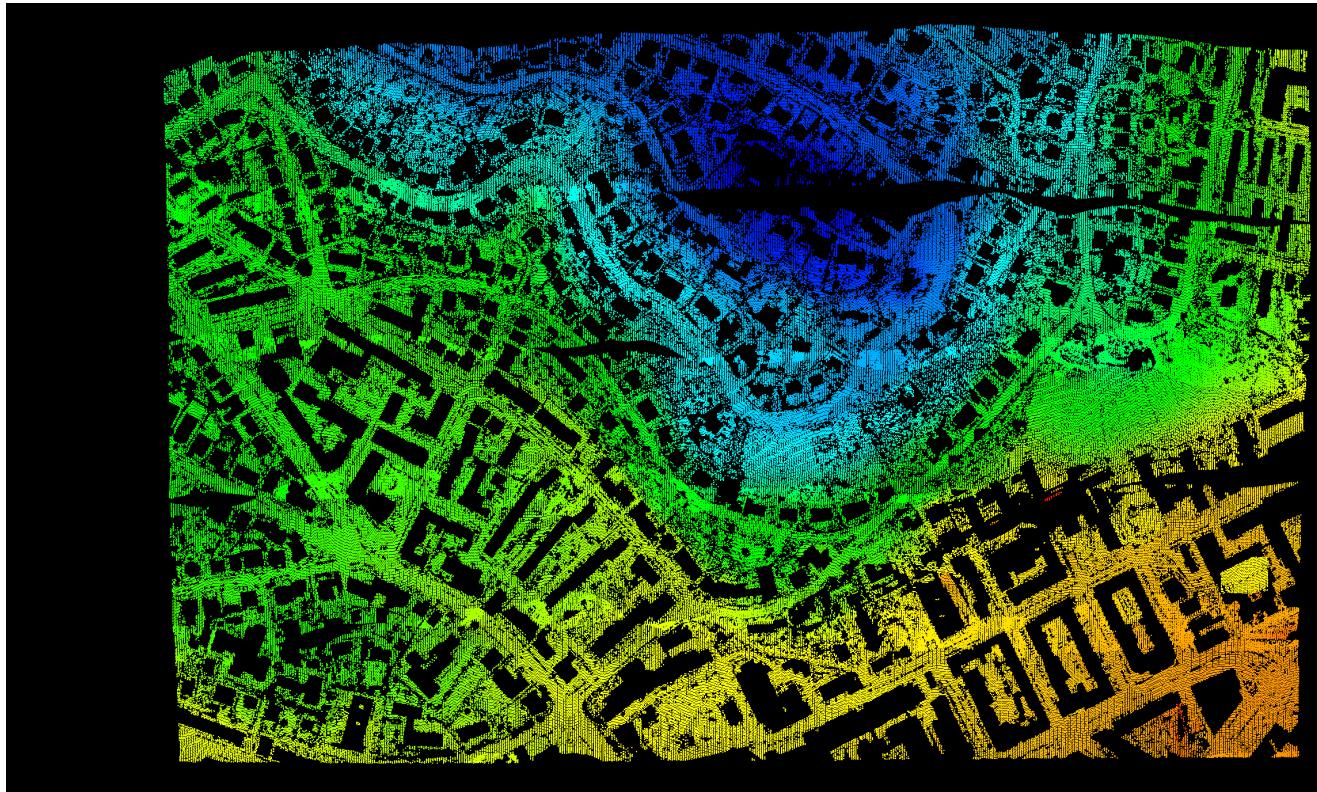
```
Step 2) ProgressiveMorphologicalFilter
```

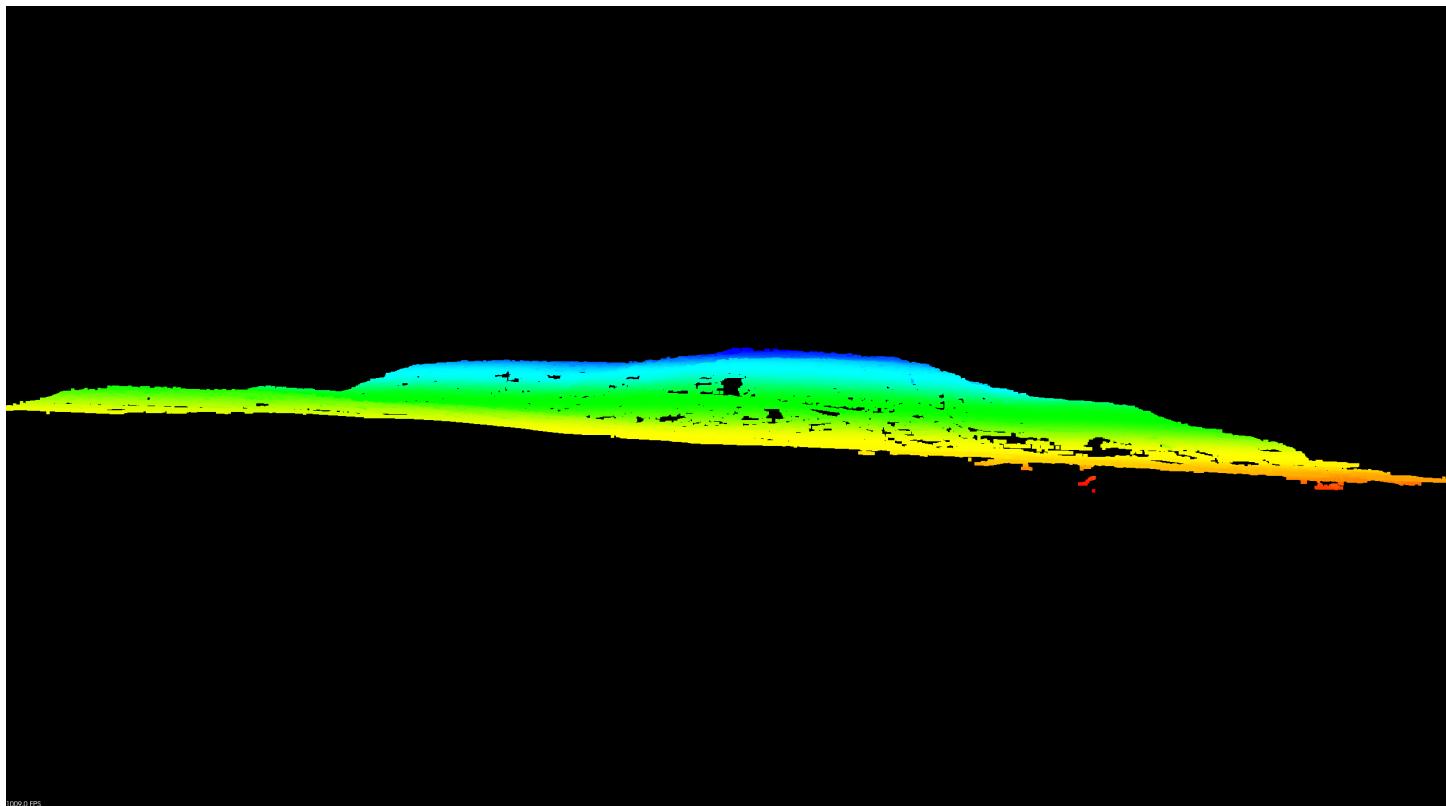
```
max window size: 33  
slope: 1.000000  
max distance: 2.500000  
initial distance: 0.150000  
cell size: 1.000000  
base: 2.000000  
exponential: true  
negative: false  
Iteration 0 (height threshold = 0.150000, window size = 3.
```

```
→000000) ...ground now has 874094 points
```

```
Iteration 1 (height threshold = 2.150000, window size = 5.  
→000000)...ground now has 837141 points  
Iteration 2 (height threshold = 2.500000, window size = 9.  
→000000)...ground now has 762213 points  
Iteration 3 (height threshold = 2.500000, window size = 17.  
→000000)...ground now has 632827 points  
Iteration 4 (height threshold = 2.500000, window size = 33.  
→000000)...ground now has 596620 points  
1356744 points filtered to 596620 following progressive  
→morphological filter
```

The result is noticeably cleaner in both the top-down and front views.





Unfortunately, you may notice that we still have a rather large building in the lower right of the image. By tweaking the parameters slightly, in this case, increasing the cell size, we can do a better job of removing such features.

```
{  
    "pipeline": {  
        "name": "Progressive Morphological Filter with Outlier Removal",  
        "version": 1.0,  
        "filters": [{  
            "name": "StatisticalOutlierRemoval",  
            "setMeanK": 8,  
            "setStddevMulThresh": 3.0  
        }, {  
            "name": "ProgressiveMorphologicalFilter",  
            "setCellSize": 1.5  
        }]  
    }  
}
```

```
$ pdal pcl -i CSite1_orig-utm.laz -o CSite1_orig-utm-ground.laz -p  
→sor-pmf2.json -v4
```

```
NAME: Progressive Morphological Filter with Outlier Removal (1.0)
HELP:
AUTHOR:
-----
→
process tile 0 through the pipeline

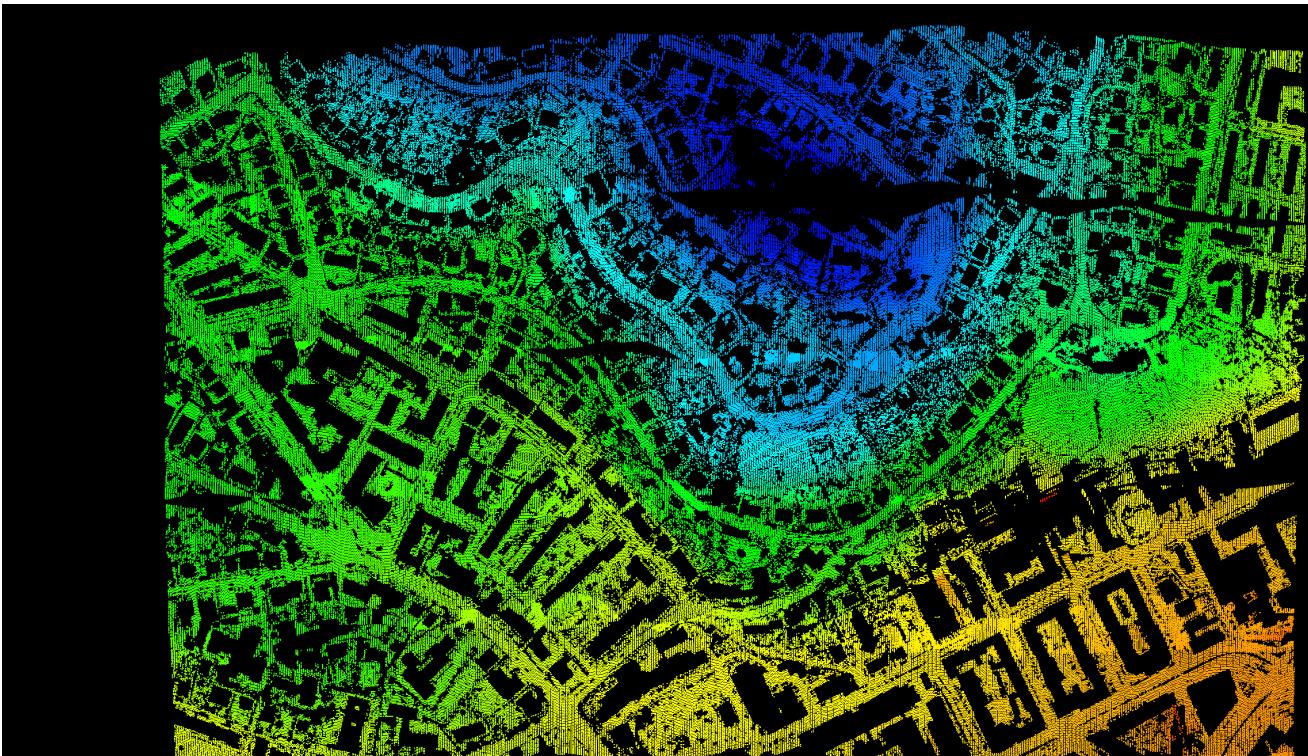
Step 1) StatisticalOutlierRemoval

    8 neighbors and 3.000000 multiplier
    1366408 points filtered to 1356744 following outlier removal

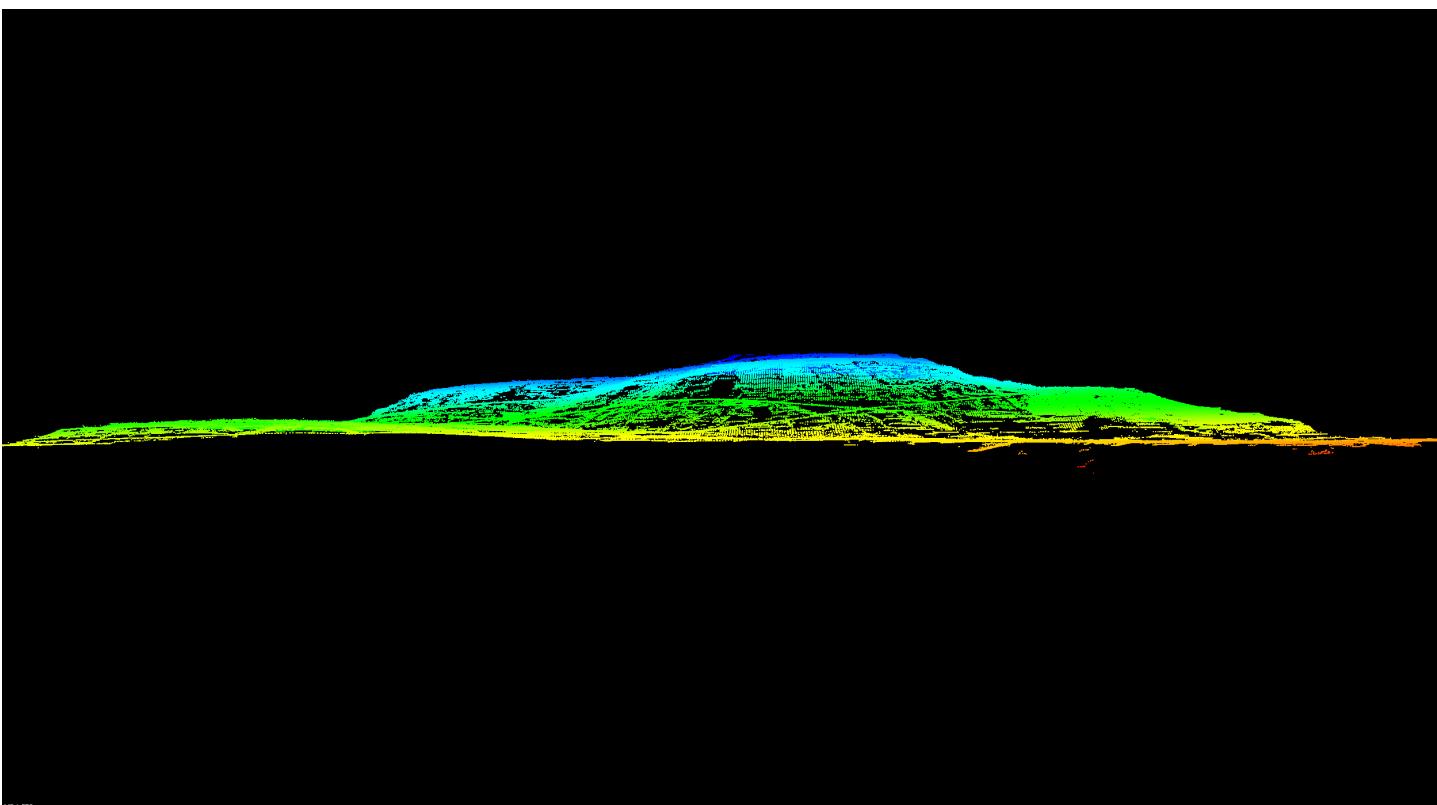
Step 2) ProgressiveMorphologicalFilter

    max window size: 33
    slope: 1.000000
    max distance: 2.500000
    initial distance: 0.150000
    cell size: 1.500000
    base: 2.000000
    exponential: true
    negative: false
    Iteration 0 (height threshold = 0.150000, window size = 4.
→500000) ...ground now has 785496 points
    Iteration 1 (height threshold = 2.500000, window size = 7.
→500000) ...ground now has 728738 points
    Iteration 2 (height threshold = 2.500000, window size = 13.
→500000) ...ground now has 623385 points
    Iteration 3 (height threshold = 2.500000, window size = 25.
→500000) ...ground now has 581679 points
    Iteration 4 (height threshold = 2.500000, window size = 49.
→500000) ...ground now has 551006 points
    1356744 points filtered to 551006 following progressive_
→morphological filter
```

Once again, the result is noticeably cleaner in both the top-down and front views.



521.5 FPS



489.0 FPS

10.1.5 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 being able to cut or clip point cloud data with 2D geometries, often from auxillary data sources such as [OGR](http://www.gdal.org) (<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* \(page 223\)](#)
 - [*Introduction* \(page 223\)](#)
 - [*Example Data* \(page 223\)](#)
 - [*Stage Operations* \(page 224\)](#)
 - [*Data Preparation* \(page 224\)](#)
 - [*Pipeline* \(page 225\)](#)
 - [*Processing* \(page 227\)](#)
 - [*Conclusion* \(page 227\)](#)

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:

- <http://www.liblas.org/samples/autzen/autzen.laz>
- <https://github.com/PDAL/PDAL/raw/master/test/data/autzen/attributes.json>

Stage Operations

This operation depends on two stages PDAL provides. The first is the [filters.overlay](#) (page 154) stage, which allows you to assign point values based on polygons read from [OGR](#) (<http://www.gdal.org>). The second is the [filters.range](#) (page 170), which allows you to keep or reject points from the set that match given criteria.

See also:

[filters.python](#) (page 164) or [filters.matlab](#) (page 144) allows you to construct sophisticated logic for keeping or rejecting points in a more expressive environment ([Python](#) (<http://www.python.org>) or ([Matlab](#) (<http://www.mathworks.com>))..

Data Preparation

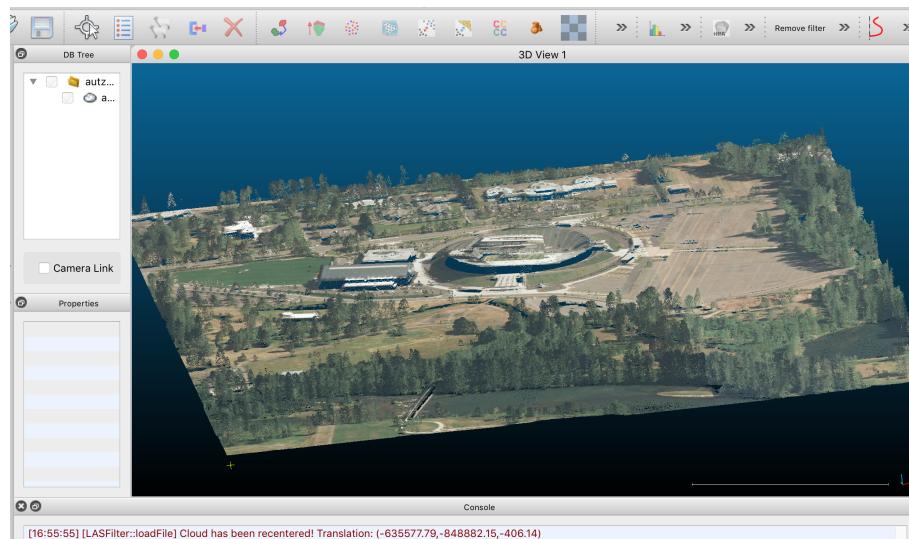


Fig. 10.1: Autzen Stadium, a 100 million+ point cloud file.

The data are mixed in two different coordinate systems. The [LAZ](#) (page 59) 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=399999.999999999 +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.

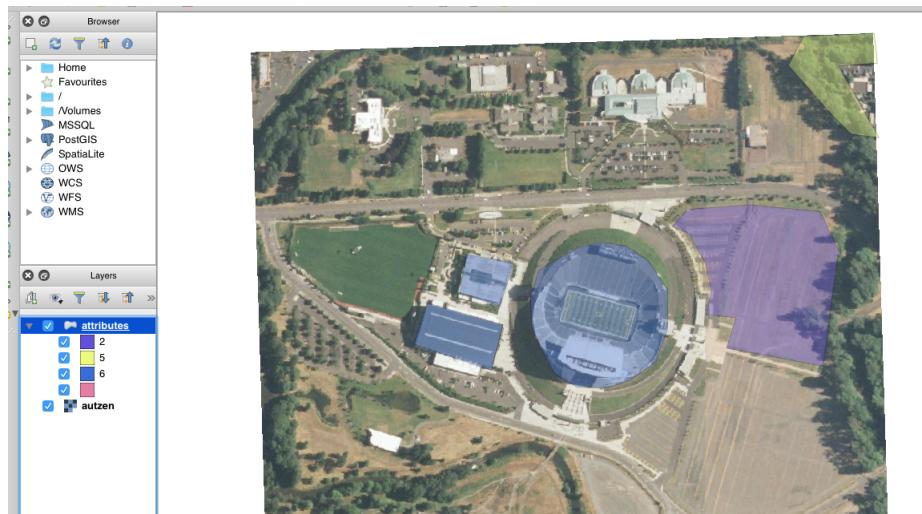


Fig. 10.2: We want to clip out the polygon in the upper right hand corner. We can view the GeoJSON (<http://geojson.org>) geometry using something like QGIS (<http://qgis.org>)

Pipeline

A PDAL *Pipeline* (page 41) is how you define a set of actions to happen to data as they are read, filtered, and written.

```
{  
  "pipeline": [  
    "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 59): 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 154): Using the VRT we defined in **Data Preparation** (page 224), read attribute polygons out of the data source and assign the values from the `CLS` column to the `Classification` field.
- *filters.range* (page 170): 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.
- *writers.las* (page 85): write our content back out using an **ASPRS LAS** (<http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html>) writer.

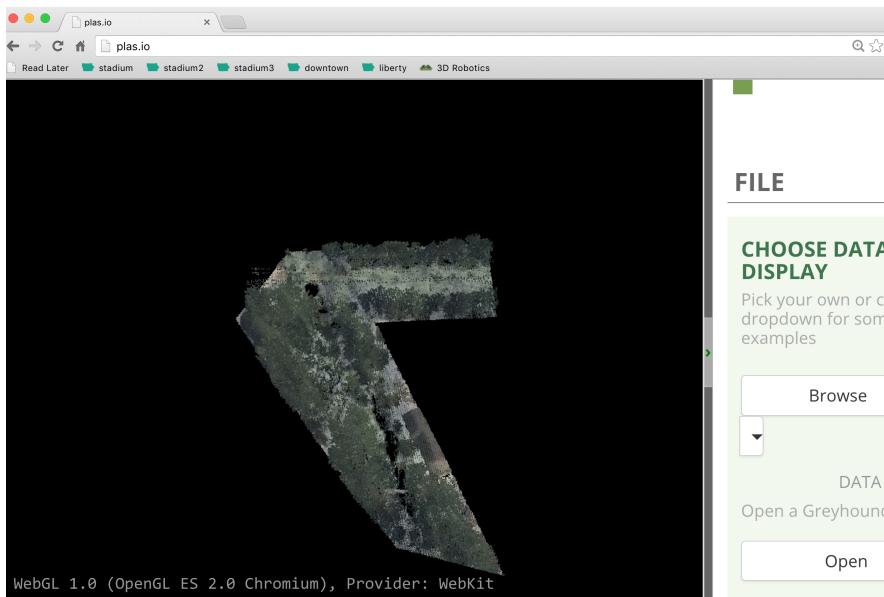
Note: You don't have to use only `Classification` to set the attributes with *filters.overlay* (page 154). Any valid dimension name could work, but most LiDAR softwares will display categorical coloring for the `Classification` field, and we can leverage that behavior in this scenario.

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. Call `pdal pipeline` on the *Pipeline* (page 41).

```
$ pdal pipeline shape-clip.json
```

3. Visualize `output.las` in an environment capable of viewing it. <http://plas.io> or [CloudCompare](http://www.danielgm.net/cc/) (<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 154) and `filters.range` (page 170) stages to clip points with shapefiles.

10.1.6 Performing Poisson Sampling of Point Clouds Using Dart Throwing

Author Bradley Chambers

Contact brad.chambers@gmail.com

Date 11/13/2015

This tutorial will describe the creation of a new filter for sampling point cloud data, `filters.sample` (page 175).

Note: `filters.dartsample` required PCL and has since been replaced by `filters.sample` (page 175), which is a native PDAL filter. We leave this tutorial as an example of how to create a filter, and of how to work with the PCL plugin, but the filter in reference is no longer available.

Introduction

Sampling of point cloud data can be advantageous for a number of reasons. It can be used with care to create a lower resolution version of the point cloud for visualization, or to accelerate processing of derivative products at a coarser resolution. “Duplicate” points can be removed by subsampling. And the effects of overlapping and uneven scan patterns can be removed by sampling.

Approach

We have chosen to implement the dart throwing filter as a PCL filter. The reason for this is simple. While it is often the case that we want to create a filter that can be included as stage in a PDAL pipeline, there are other, more primitive filters, that we may wish to reuse within another filter. While we may not want to subsample our point cloud, thus discarding points from our output `PointView`, we may be perfectly content to subsample the data as a transient representation of the data within a filter - for example, to compute a coarse estimate of ground returns on 10% of the data to accelerate detection of buildings at full resolution.

Our implementation is a no frills, brute force approach (there are more advanced methods that have been published over the years).

We begin by creating a random permutation of the input `PointView` indices. The first point in this randomized set is appended to the output `PointView` and its associated octree. For the remaining points in the input cloud, we first test to see if there is a neighbor in the output cloud within the specified `radius`. If so, we discard the point. If not, it too is added to the output `PointView` and octree.

That's it! It's really that simple.

Example #1

We will again be working with the autzen dataset. The dart sampling filter is easily invoked via the PDAL `translate` command. Here, we enforce a minimum distance of three feet.

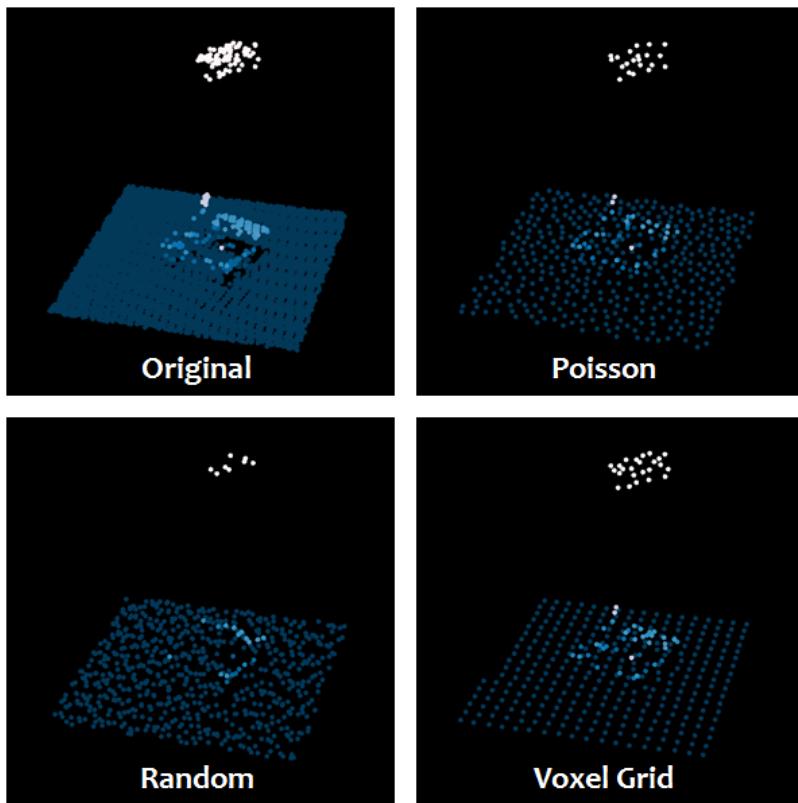
```
$ pdal translate autzen.laz autzen-dart-sampled.laz dartsample \
--filters.dartsample.radius=3
```

For comparison, we will process autzen using both [filters.decimation](#) (page 119), with a step size of 6, and [filters.voxelgrid](#) (page 184), with a leaf size of 4.5 feet in X, Y, and Z, to arrive at a subsampled point cloud of roughly 1.6 million points, which closely approximates the number of points returned in our dart sampling run.

```
$ pdal translate autzen.laz autzen-randomly-sampled.laz decimation \
--filters.decimation.step=6
```

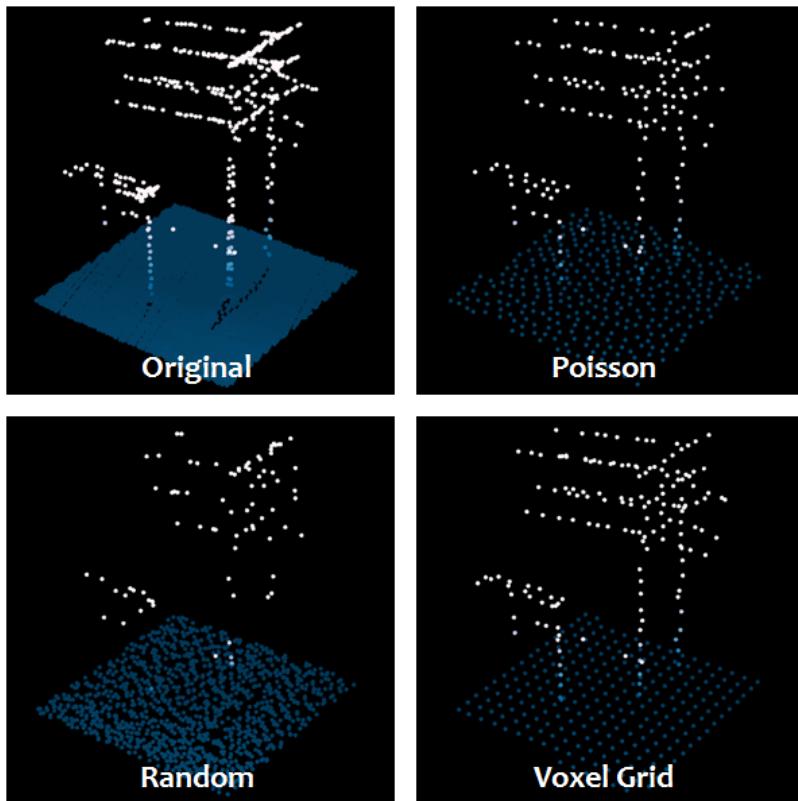
```
$ pdal translate autzen.laz autzen-voxelgrid-sampled.laz voxelgrid \
--filters.voxelgrid.leaf_x=4.5 --filters.voxelgrid.leaf_y=4.5 --
--filters.voxelgrid.leaf_z=4.5
```

First, we inspect a fixture located in the middle of an open field, displaying the original point cloud, Poisson (dart sampled), voxel grid, and randomly sampled results in clockwise order from top-left.

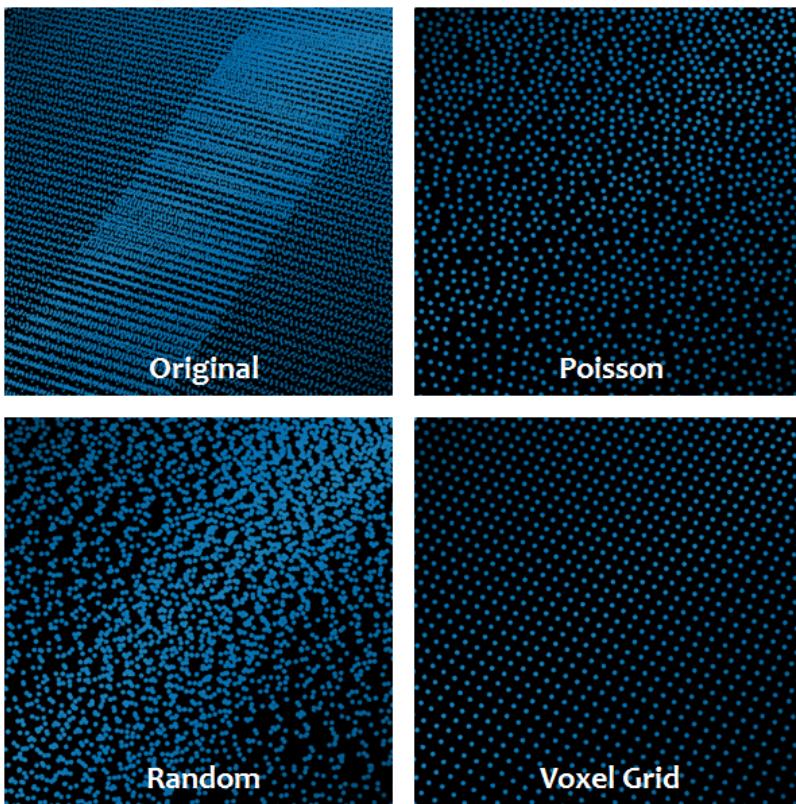


The Poisson and voxel gridded clouds both appear to do a good job retaining key elements of the structure, while significantly reducing the number of points allocated to the ground plane. The randomly sampled cloud however, exhibits the opposite behavior, noticeably degrading the structure, while heavily sampling the ground. This is not an unexpected result, as both the Poisson and voxel grid approaches take the data and its distribution into account during the subsampling process, while random sampling considers only point order (keep every N-th point).

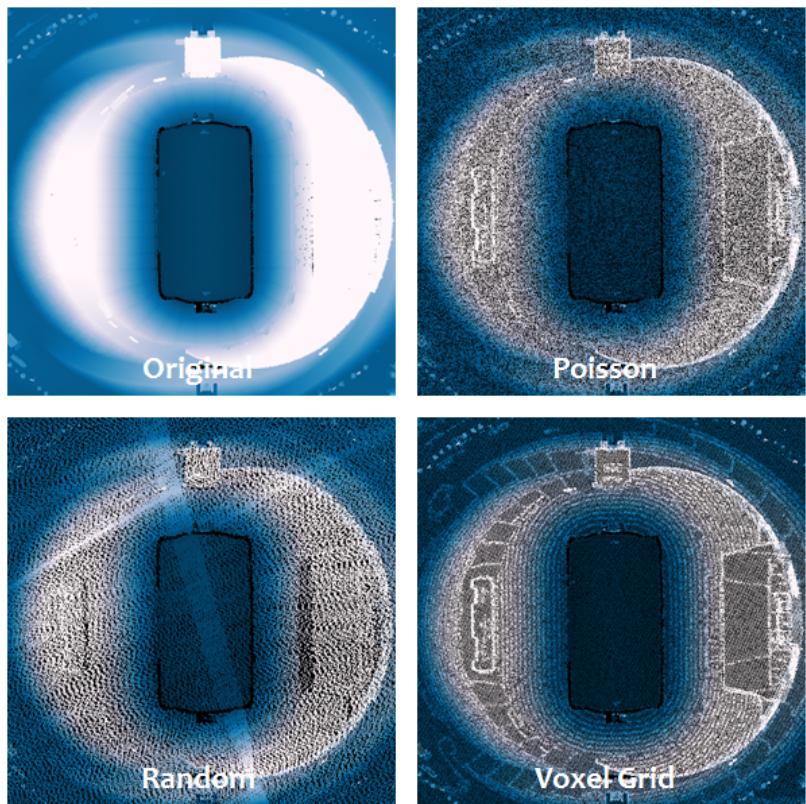
The second example shows a very similar result, this time with a set of point pylons and power lines. The random sampling approach severely degrades the structures in the scene, while the Poisson and voxel grid techniques both preserve signal.



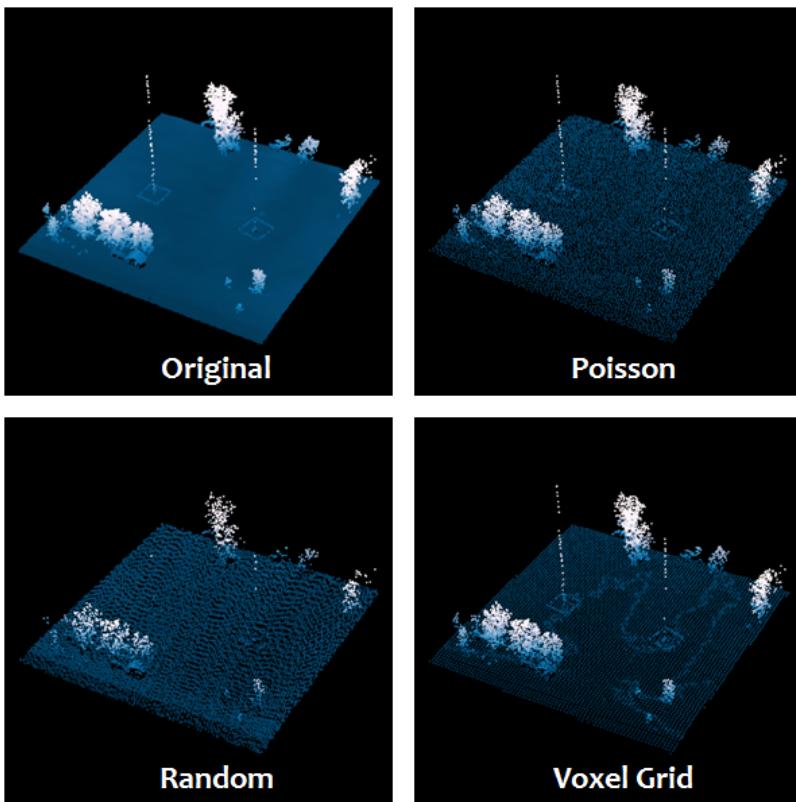
In the next example, we can see that there is an increased number of points in a scan overlap region. This is not uncommon, as data collectors strive to avoid gaps in coverage and overlap datasets to aid in registration of multiple passes. The appearance of these denser regions can be distracting to the eye, and the Poisson and voxel grid subsampling method can both be used to make the collected points appear more uniform by culling those points that are very near other other points. The random sampling method preserves this artifact.



In this top-down view of a football stadium, we again see that the random sampling technique preserves (and perhaps even accentuates) scan pattern and overlap artifacts. It also introduces a side effect to the voxel grid approach, an aliasing of the data, seen as staircasing in the sloping surfaces of the stadium.



Our last example once again demonstrates each of the issues we have identified. The random sampling result eliminates a majority of points from each of the towers and highlights a scan overlap region. The voxel grid method results in ringing in the sloped terrain. The Poisson approach preserves a good amount of detail in the original signal and does not introduce any visual artifacts.



10.1.7 Filtering Data with Python

Author Howard Butler

Contact howard@hobu.co

Date 5/12/2017

Table of Contents

- *Filtering Data with Python* (page 233)
 - *Introduction* (page 234)
 - *The Challenge* (page 234)
 - *Python Filter* (page 235)
 - *PDAL Pipeline* (page 235)
 - *Docker* (page 236)
 - *Final Script* (page 236)

This tutorial will describe using [*filters.python*](#) (page 164) to identify outlier points in an LAS

file.

Introduction

Noise filtering is a primary challenge in point cloud processing. There are many stock options available to you in PDAL to achieve it. These include:

- *filters.iqr* (page 139)
- *filters.elm* (page 122)
- *filters.outlier* (page 151)

These filters remove points that are deemed outliers in different ways. The don't, however, identify specific points. Python plus [NumPy](http://www.numpy.org/) (<http://www.numpy.org/>) would be a very quick way to prototype a tool that identified specific points we would like to filter.

PDAL has three different ways to manipulate data with Python. The first is *filters.python* (page 164), which we will be using in this tutorial. The second is the Python extension at <https://pypi.python.org/pypi/PDAL> that allows you to utilize PDAL processing operations in your own Python programs.

See also:

[Python](#) (page 189) describes PDAL's Python story in more detail.

The Challenge

We have a ASPRS LAS

(<http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html>) file that has some points that are completely out of range and have bad Y values. We don't know how those points got in the file, but we want to identify **which** points they were, because it might be some indication about a failure point in our processing operation. We could simply use *filters.mad* (page 143) to remove these with stock PDAL operations, but that filter doesn't tell us which points it removed. Our own custom Python filter can do this for us.

Processing Strategy

The technique that *filters.mad* (page 143) uses will suit our problem. In short, we want to identify which points have Y values that are more than three standard deviations away from the median. We are going to print a [JSON](#) (<https://en.wikipedia.org/wiki/JSON>) object that will contain our information, which we can then use in some downstream processing operation (as filtered with the [jq](#) (<https://stedolan.github.io/jq/>) command line utility).

We also want to use [Install Docker](#) (page 15) to process our data with a [Bash](#) ([https://en.wikipedia.org/wiki/Bash_\(Unix_shell\)](https://en.wikipedia.org/wiki/Bash_(Unix_shell))) script. We want to know for sure that we're

using PDAL's standard release (1.5 in this case), and we'd like for our operation to be agnostic to the platform on which it is running.

Python Filter

Through the use of the [filters.python](#) (page 164), PDAL allows the use of Python and NumPy (<http://www.numpy.org/>) to process point cloud data. This can be very useful in prototyping situations, where PDAL can provide convenient data access and the processing logic of software that is still taking shape can be constructed with the rapid prototyping tools that Python can provide.

```
1 import numpy as np
2
3 def mad(ins, outs):
4
5     # Fetch the Y dimension, which has our outliers
6     y = ins["Y"]
7
8     # Let numpy compute median
9     median = np.median(y)
10
11    # Let numpy compute stddev
12    stddev = np.std(y)
13
14    # Identify points that are > 3 stddev
15    indexes = np.where( abs(y-median) > 3*stddev )[0]
16
17    # Stuff our output in a dictionary
18    output = {}
19    output['median'] = median
20    output['stddev'] = stddev
21    output['indexes'] = list([v for v in indexes])
22    output['values'] = [y[i] for i in indexes]
23
24    # Print our dict to stdout
25    print output
26
27    # filters.python must return True to tell
28    # PDAL it successfully completed
29    return True
```

PDAL Pipeline

Our pipeline to apply the filter is straightforward. Because we are going to be processing the data with a Bash script, we'll use some substitution to push our filename (taken from an

argument on the command line) and our script itself. There is nothing much interesting except for the fact that we substitute our filename in its own entry so PDAL can figure out which driver to read it with. This will allow our script to work with any format PDAL can identify.

```
1  {
2    "pipeline": [
3      "/data/$filename",
4      {
5        "type" : "filters.python",
6        "function": "mad",
7        "module": "anything",
8        "source": "$script"
9      }
10 }
```

Docker

As I mentioned before, we want to run this script using Docker so we have consistent versioning and so we can use the script in the same way across a bunch of machines. To run the command in Docker we need to do two things:

1. Start a Docker container with the PDAL Docker container tag (pdal/pdal:1.5 in this case)
2. Run our command on that image once it is running.

docker run outputs the SHA of the container instance it is running to STDIN. We will capture that and then run our pdal pipeline command on it.

```
# Start the container
container=$(docker run -it -v `pwd`:/data pdal/pdal:1.5)

# Echo our pipeline into it and run pdal pipeline on it
output=$(echo $pipeline | docker exec -i $container pdal pipeline -i _  
→STDIN)

# redirect stderr and stdout to null. We don't want to know which ID
# was killed.
docker kill $container &> /dev/null
```

Final Script

Here's our final script. The junk from lines 53-66 are to get bash to do variable substitution into our *Pipeline* (page 41) **JSON** (<https://en.wikipedia.org/wiki/JSON>) without breaking its syntax doing double quote and newline substitution.

Using our script, we can now identify which points have Y dimensions greater than three standard deviations:

```
$ ./run.sh bad-y-values.las
{'values': [22940882.882926211, 18747910.56323766], 'median': ↵
↪ 4322652.908885533, 'stddev': 8292.74344366484, 'indexes': [1375302,
↪ 1376129] }
```

```
1 #!/bin/bash
2
3 filename="$1"
4
5 read -d '' script <<"EOF"
6
7 import numpy as np
8
9 def mad(ins, outs):
10
11     # Fetch the Y dimension, which has our outliers
12     y = ins["Y"]
13
14     # Let numpy compute median
15     median = np.median(y)
16
17     # Let numpy compute stddev
18     stddev = np.std(y)
19
20     # Identify points that are > 3 stddev
21     indexes = np.where( abs(y-median) > 3*stddev )[0]
22
23     # Stuff our output in a dictionary
24     output = {}
25     output['median'] = median
26     output['stddev'] = stddev
27     output['indexes'] = list([v for v in indexes])
28     output['values'] = [y[i] for i in indexes]
29
30     # Print our dict to stdout
31     print output
32
33     # filters.python must return True to tell
34     # PDAL it successfully completed
35     return True
36
37 EOF
38
39 read -d '' pipeline <<"EOF"
```

```
40 {
41     "pipeline": [
42         "/data/$filename",
43         {
44             "type" : "filters.python",
45             "function": "mad",
46             "module": "anything",
47             "source": "$script"
48         }
49     ]
50 EOF
51
52
53 # Do a bunch of bash vomit to properly substitute
54 # in our $filename and $script variables into the pipeline
55 # variable
56 IFS=%"
57
58 # echo newlines
59 script=$(awk '{printf "%s\\n", $0}' <<< "$script")
60
61 # replace " with \
62 script=$(sed 's/"/\\\"/g' <<< "$script")
63
64 pipeline=$(awk '{printf "%s", $0}' <<< "$pipeline")
65 pipeline=$(sed 's/"/\\\"/g' <<< "$pipeline")
66 pipeline=$(eval echo $pipeline)
67
68
69 # Start the container
70 container=$(docker run -it -v `pwd`:/data pdal/pdal:1.5)
71
72 # Echo our pipeline into it and run pdal pipeline on it
73
74 output=$(echo $pipeline | docker exec -i $container pdal pipeline -i<
75 ↴STDIN)
76
77 # redirect stderr and stdout to null. We don't want to know which ID
78 # was killed.
79 docker kill $container &> /dev/null
```

CHAPTER
ELEVEN

WORKSHOP

11.1 Point Cloud Processing and Analysis with PDAL

Author Howard Butler

Author Pete Gadowski

Author Dr. Craig Glennie

Contact howard@hobu.co

Date 07/25/2017

11.1.1 Introduction

1. *Introduction to LiDAR* (page 240)
2. *Introduction to PDAL* (page 5)
3. *Software Installation* (page 246)
4. *Basic Information* (page 248)
5. *Translation* (page 255)
6. *Analysis* (page 264)
7. *Georeferencing* (page 313)

Materials

Slides

- Slides (<https://www.pdal.io/workshop/slides/>)

Workshop Materials

These materials are available at <http://pdal.io/workshop/> as both a PDF and an HTML website.

- [PDF download](https://pdal.io/PDAL.pdf) (<https://pdal.io/PDAL.pdf>)
- [Website](https://pdal.io/workshop/) (<https://pdal.io/workshop/>)

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
<https://s3.amazonaws.com/pdal/workshop/PDAL.zip>

11.1.2 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 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 [\[Gle07\]](#) (page 457):

$$\mathbf{p}_G^l = \mathbf{p}_{GPS}^l + \mathbf{R}_b^l (\mathbf{R}_s^b \mathbf{r}^s - \mathbf{l}^b) \quad (11.1)$$

where:

- \mathbf{p}_G^l are the coordinates of the target point in the global reference frame
- \mathbf{p}_{GPS}^l are the coordinates of the GNSS sensor in the global reference frame
- \mathbf{R}_b^l is the rotation matrix from the navigation frame to the global reference frame
- \mathbf{R}_s^b is the rotation matrix from the scanner's frame to the navigation frame (boresight matrix)
- \mathbf{r}^s is the coordinates of the laser point in the scanner's frame
- \mathbf{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 (11.1) 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 analyse or process full waveform data.

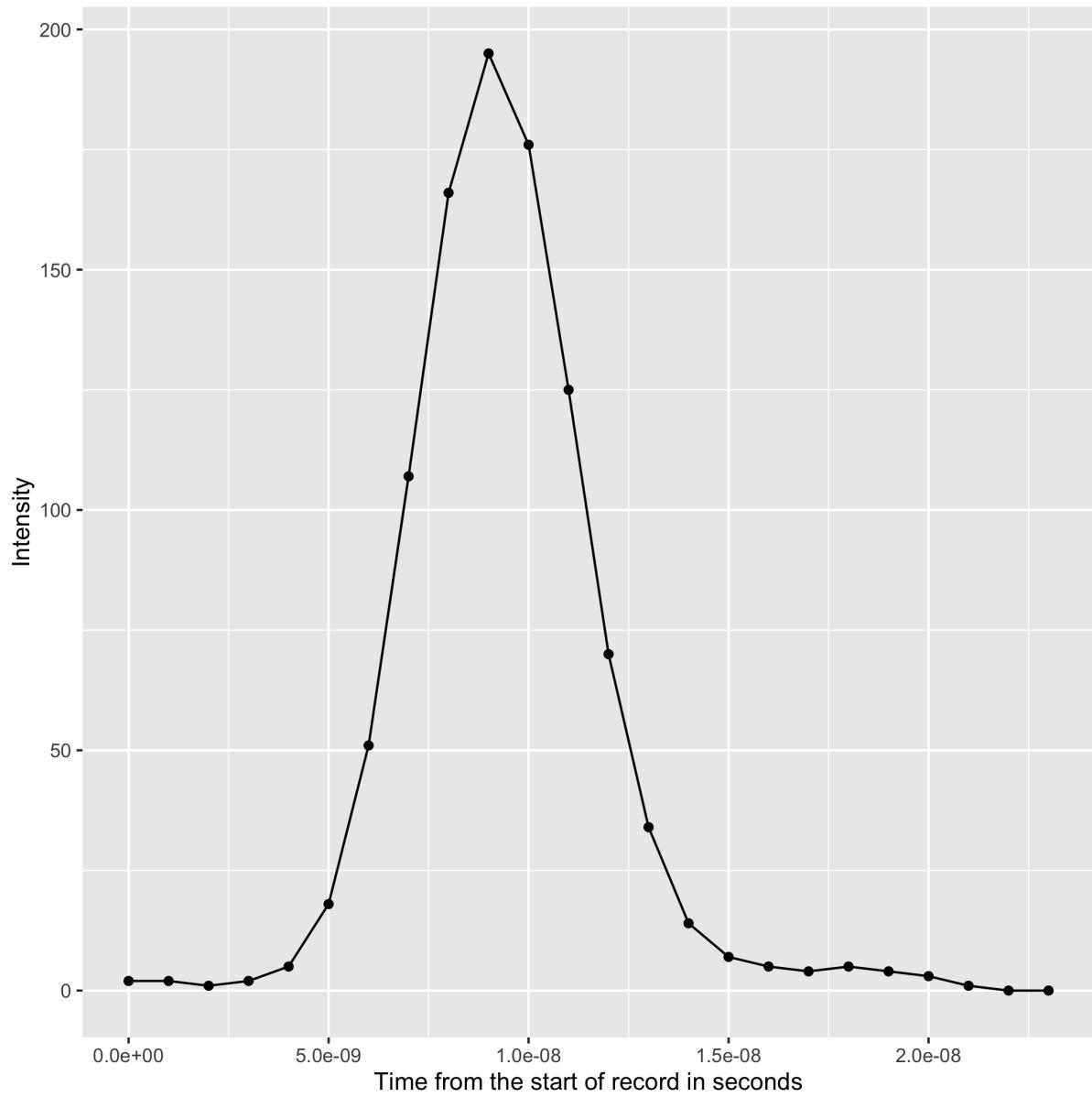


Fig. 11.1: A real-world outgoing LiDAR pulse.

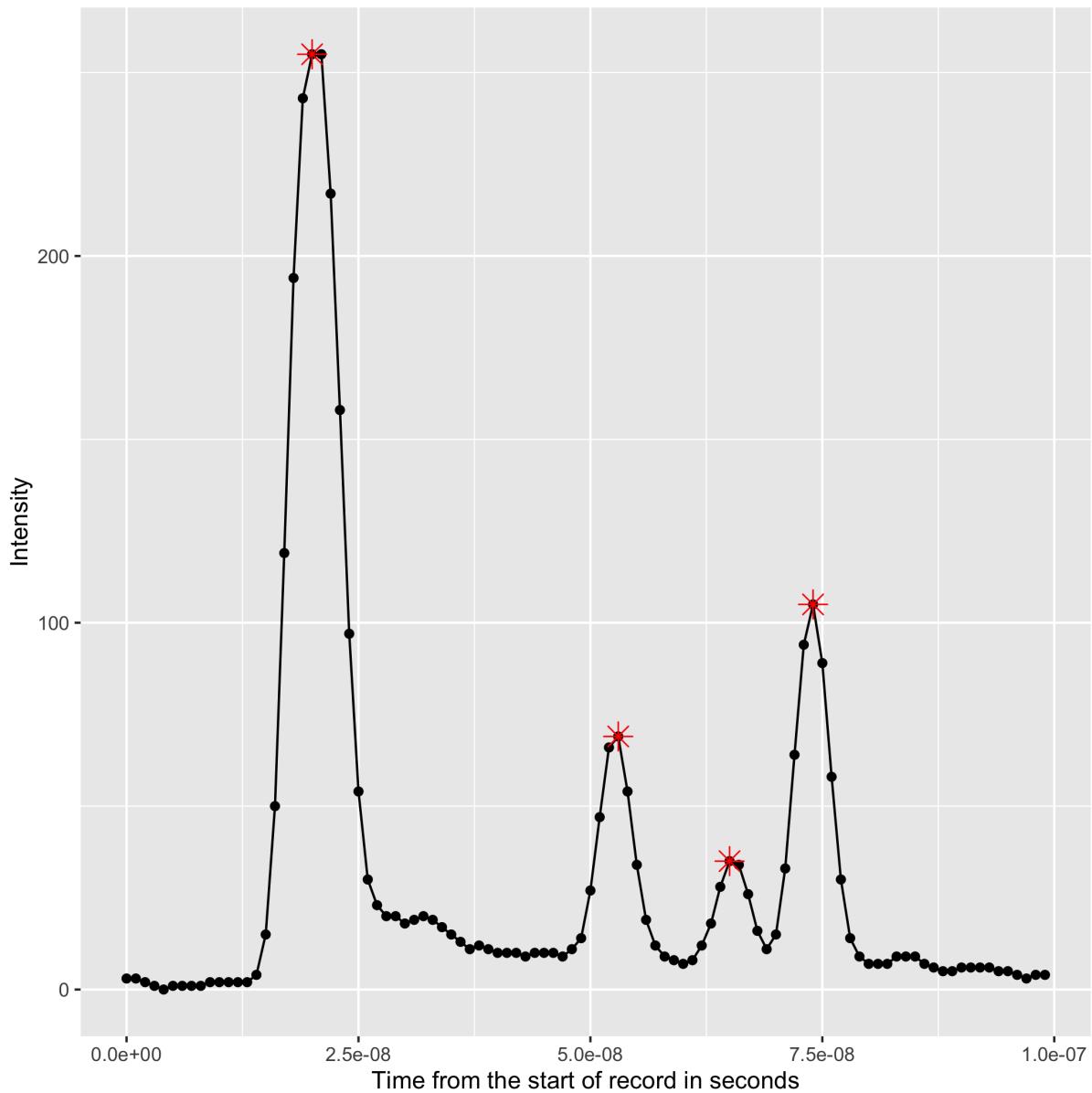


Fig. 11.2: A real-world incoming LiDAR return. Potential discrete-return peaks are marked in red.

11.1.3 Software Installation

OSGeo4W

What is OSGeo4W?

OSGeo4W (<https://trac.osgeo.org/osgeo4w/>) is a distribution of geospatial software built for Windows. The PDAL project provides Windows builds through this distribution.

How will we use OSGeo4W?

PDAL stands on the shoulders of giants. It uses GDAL, GEOS, and *many other dependencies* (page 334). Because of this, it is very challenging to build it yourself. We could easily burn an entire workshop learning the esoteric build miseries of PDAL and all of its dependencies. Fortunately, OSGeo4W provides us a fully-featured known configuration to run our examples and exercises without having to suffer so much.

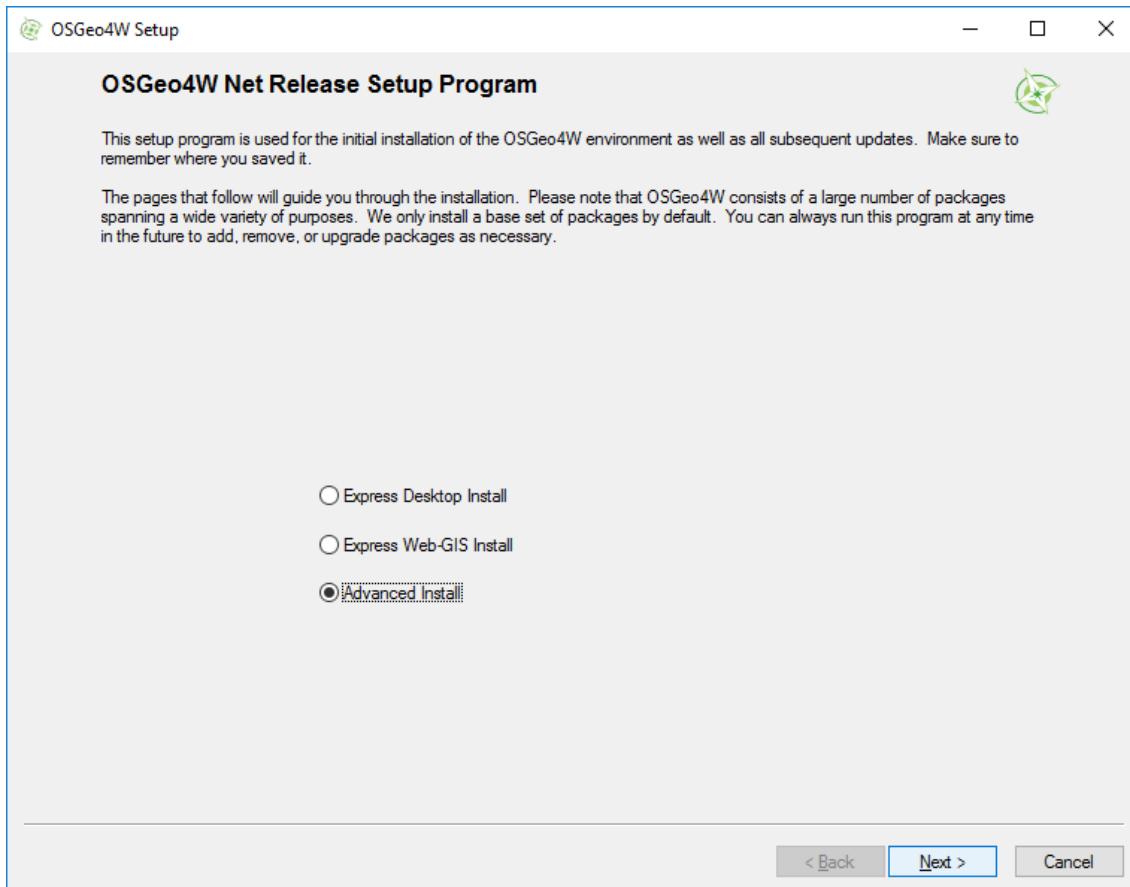
Note: Not everyone uses Windows. Another alternative to get a known configuration is to go through the workshop using *Install Docker* (page 15) as your platform. A previous edition of the workshop was provided as Docker, but it was found to be a bit too difficult to follow.

Installing OSGeo4W

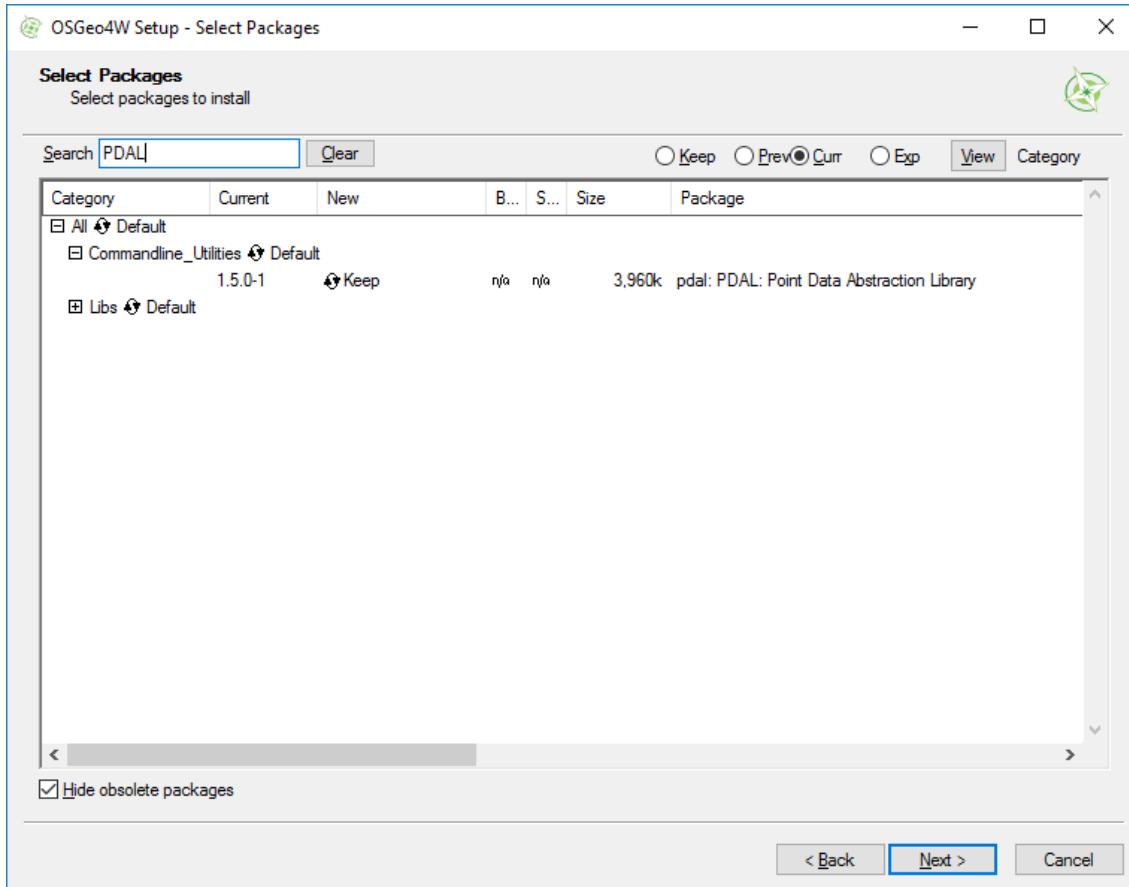
1. Copy the entire contents of your workshop USB key to a PDAL directory in your home directory (something like C:\Users\hobu\PDAL). We will refer to this location for the rest of the workshop materials.
2. Download the OSGeo4W setup.exe
http://download.osgeo.org/osgeo4w/osgeo4w-setup-x86_64.exe

Warning: PDAL is only available via the 64-bit verison of OSGeo4W. There is an older 32-bit version, but none of the workshop materials will work in that environment. You must be able to run 64-bit Windows applications to follow this workshop.

3. Run the installer and choose the “Advanced Install” option.



4. Search for “PDAL” in the search box or drill down through the Commandline_Utilities section and choose the PDAL package.



11.1.4 Exercises

Basic Information

Printing the first point

Exercise

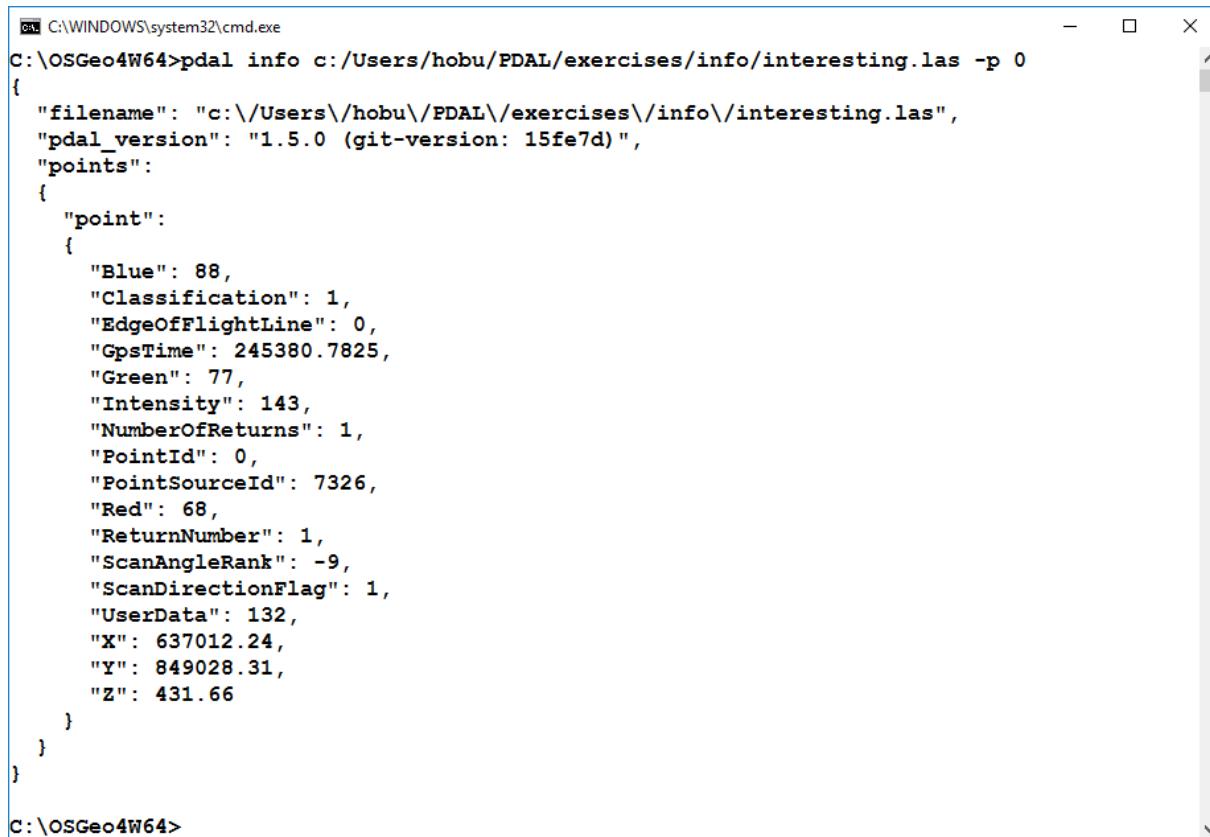
This exercise uses PDAL to print information from the first point. Issue the following command in your *OSGeo4W Shell*.

```
1 pdal info c:/Users/hobu/PDAL/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 27) on the data. All commands are run by the pdal application.

3. c:/Users/hobu/PDAL/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 31) to override which *reader* (page 50) 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).



```
C:\WINDOWS\system32\cmd.exe
C:\OSGeo4W64>pdal info c:/Users/hobu/PDAL/exercises/info/interesting.las -p 0
{
  "filename": "c:/Users/hobu/PDAL/exercises/info/interesting.las",
  "pdal_version": "1.5.0 (git-version: 15fe7d)",
  "points":
  {
    "point":
    {
      "Blue": 88,
      "Classification": 1,
      "EdgeofFlightLine": 0,
      "GpsTime": 245380.7825,
      "Green": 77,
      "Intensity": 143,
      "NumberOfReturns": 1,
      "PointId": 0,
      "PointSourceId": 7326,
      "Red": 68,
      "ReturnNumber": 1,
      "ScanAngleRank": -9,
      "ScanDirectionFlag": 1,
      "UserData": 132,
      "X": 637012.24,
      "Y": 849028.31,
      "Z": 431.66
    }
  }
}

C:\OSGeo4W64>
```

Notes

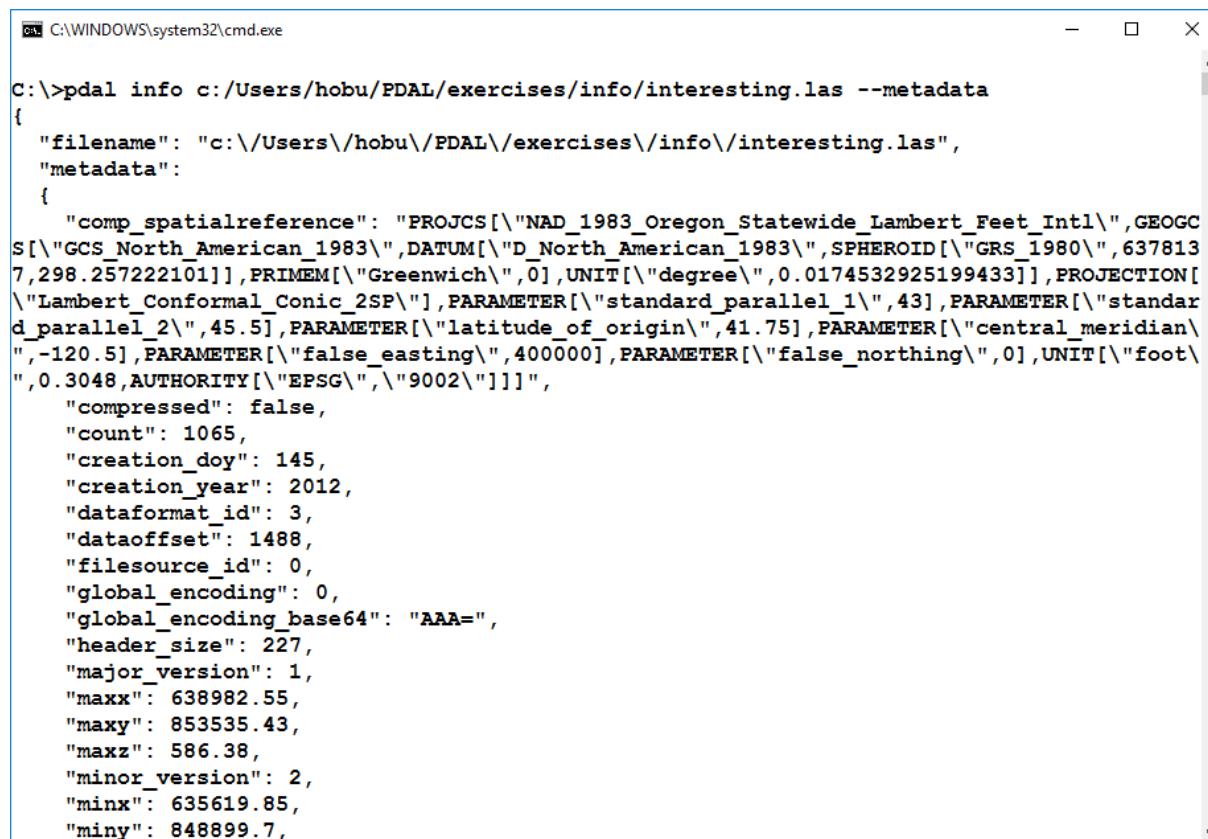
1. PDAL uses **JSON** (<https://en.wikipedia.org/wiki/JSON>) as the exchange format when printing information from *info* (page 27). JSON is a structured, human-readable format that is much simpler than its **XML** (<https://en.wikipedia.org/wiki/XML>) cousin.
2. You can use the *writers.text* (page 103) 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 *OSGeo4W Shell*.

```
1 pdal info c:/Users/hobu/PDAL/exercises/info/interesting.las --  
→metadata
```



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is 'pdal info c:/Users/hobu/PDAL/exercises/info/interesting.las --metadata'. The output is a JSON object representing the point cloud's metadata. The JSON structure includes fields like 'filename', 'comp_spatialreference', 'compressed', 'count', 'creation_doy', 'creation_year', 'dataformat_id', 'dataoffset', 'filesource_id', 'global_encoding', 'global_encoding_base64', 'header_size', 'major_version', 'maxx', 'maxy', 'maxz', 'minor_version', 'minx', and 'miny'. The 'comp_spatialreference' field is particularly complex, detailing the projection parameters.

```
C:\>pdal info c:/Users/hobu/PDAL/exercises/info/interesting.las --metadata
{
  "filename": "c:/Users/hobu/PDAL/exercises/info/interesting.las",
  "metadata": {
    "comp_spatialreference": "PROJCS[\"NAD_1983_Oregon_Statewide_Lambert_Feet_Intl\",GEOGCS[\"GCS_North_American_1983\",DATUM[\"D_North_American_1983\",SPHEROID[\"GRS_1980\",6378137,298.257222101]],PRIMEM[\"Greenwich\",0],UNIT[\"degree\",0.0174532925199433]],PROJECTION[\"Lambert_Conformal_Conic_2SP\"],PARAMETER[\"standard_parallel_1\",43],PARAMETER[\"standard_parallel_2\",45.5],PARAMETER[\"latitude_of_origin\",41.75],PARAMETER[\"central_meridian\",-120.5],PARAMETER[\"false_easting\",400000],PARAMETER[\"false_northing\",0],UNIT[\"foot\",0.3048,AUTHORITY[\"EPSG\",\"9002\"]]]",
    "compressed": false,
    "count": 1065,
    "creation_doy": 145,
    "creation_year": 2012,
    "dataformat_id": 3,
    "dataoffset": 1488,
    "filesource_id": 0,
    "global_encoding": 0,
    "global_encoding_base64": "AAA=",
    "header_size": 227,
    "major_version": 1,
    "maxx": 638982.55,
    "maxy": 853535.43,
    "maxz": 586.38,
    "minor_version": 2,
    "minx": 635619.85,
    "miny": 848899.7,
    "minz": 0
  }
}
```

Note: PDAL *metadata* (page 353) is returned in a tree structure corresponding to processing pipeline that produced it.

See also:

Use the [JSON](https://en.wikipedia.org/wiki/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) (<https://docs.python.org/2/library/json.html>)
- [jsawk](https://github.com/micha/jsawk) (<https://github.com/micha/jsawk>) (like awk but for JSON data)
- [jq](https://stedolan.github.io/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.

```
1 pdal info c:/Users/hobu/PDAL/exercises/info/interesting.las --  
  ↵ metadata ^  
2   | jq ".metadata.compressed, .metadata.system_id"
```



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is 'pdal info c:/Users/hobu/PDAL/exercises/info/interesting.las --metadata ^ More? | jq ".metadata.compressed, .metadata.system_id"'. The output shows 'false' and '"HOBU-SYSTEMID"'. The prompt 'C:\>' is visible at the bottom.

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) (<https://en.wikipedia.org/wiki/JSON>) as the exchange format when printing information from [info](#) (page 27). JSON is a structured, human-readable format that is much simpler than its [XML](https://en.wikipedia.org/wiki/XML) (<https://en.wikipedia.org/wiki/XML>) cousin.
2. The PDAL [metadata document](#) (page 353) 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 50) produce same-named values where possible, but it is common that variables are different. [Filters](#) (page 104) and even [writers](#) (page 79) can also produce metadata entries.

4. Spatial reference system or coordinate system information is a kind of special metadata, and is treated as something primary to a Stage in PDAL.

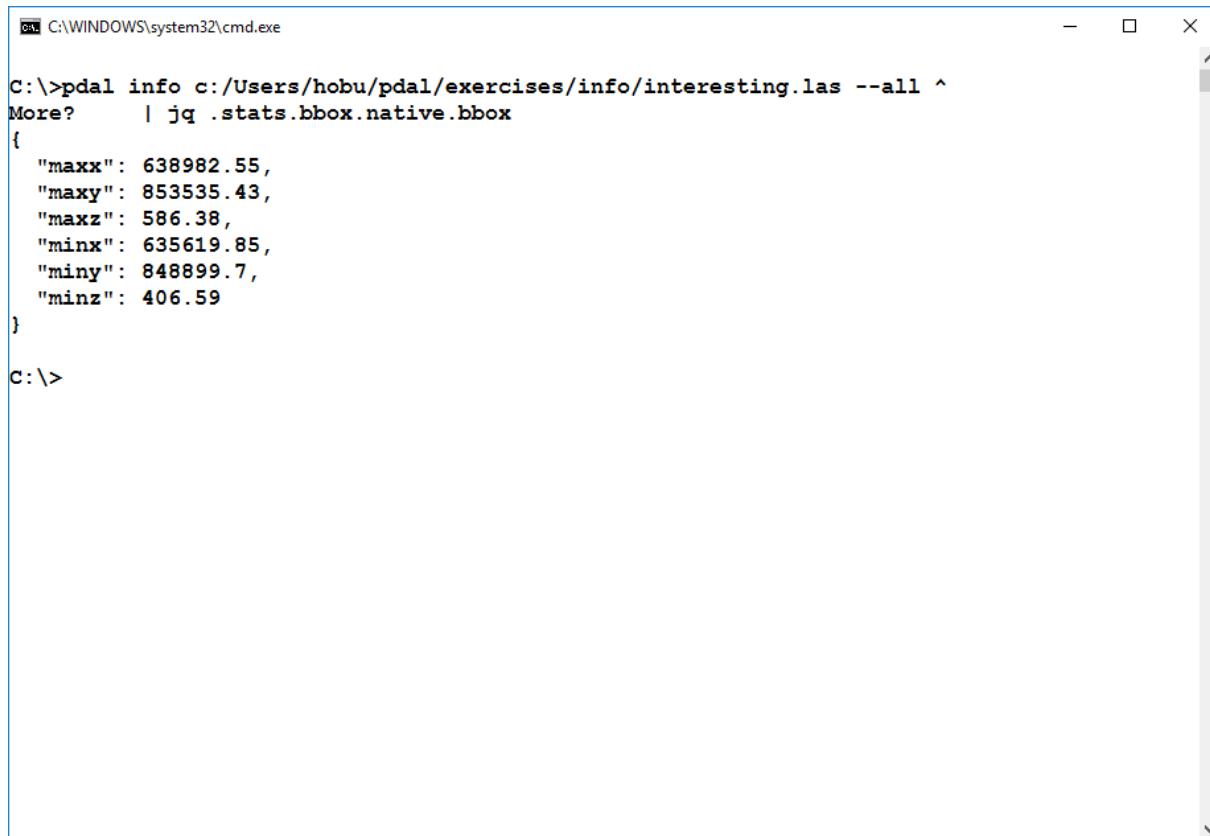
Searching near a point

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 c:/Users/hobu/pdal/exercises/info/interesting.las --all ^
| jq .stats.bbox.native.bbox
```



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The command entered is "pdal info c:/Users/hobu/pdal/exercises/info/interesting.las --all ^ | jq .stats.bbox.native.bbox". The output displayed is a JSON object representing the bounding box of the point cloud:

```
C:\>pdal info c:/Users/hobu/pdal/exercises/info/interesting.las --all ^
More?      | jq .stats.bbox.native.bbox
{
  "maxx": 638982.55,
  "maxy": 853535.43,
  "maxz": 586.38,
  "minx": 635619.85,
  "miny": 848899.7,
  "minz": 406.59
}
```

The command prompt then shows the prompt "C:\>" again.

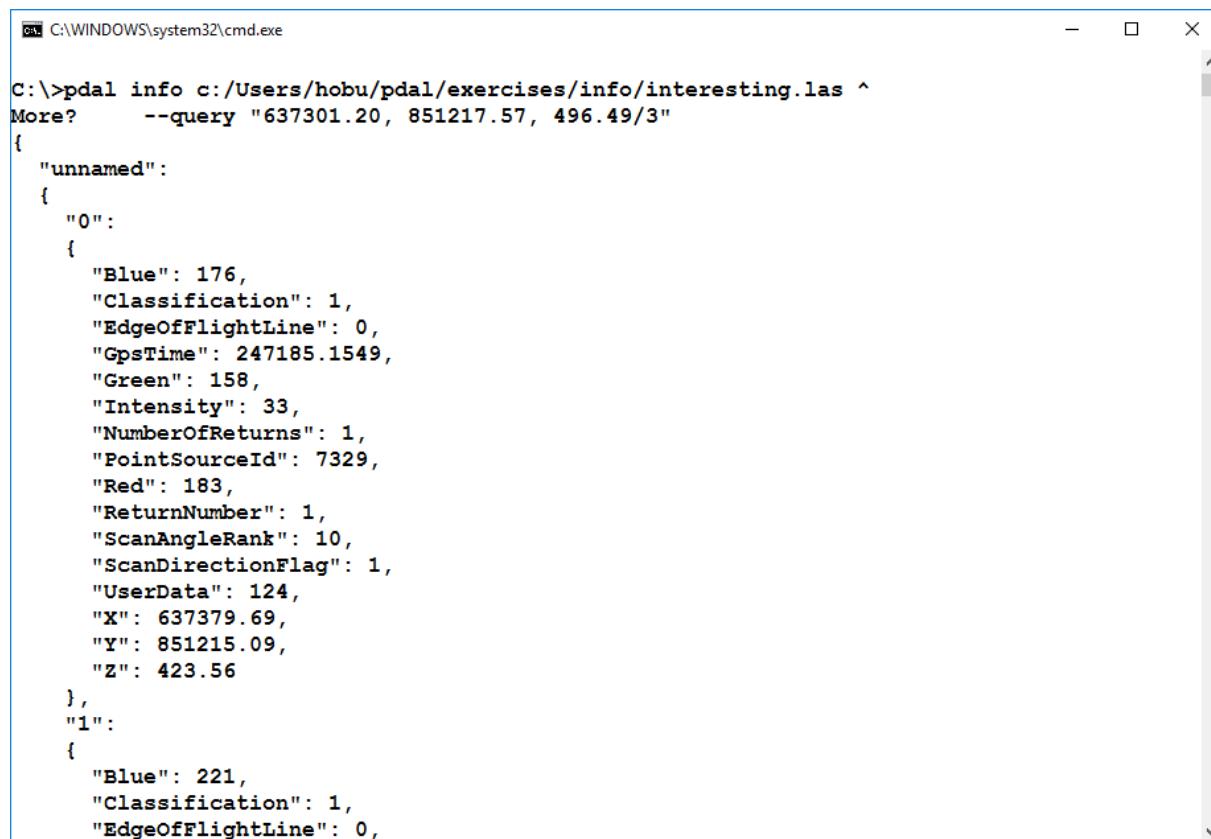
Find the average the X, Y, and Z values:

```
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 c:/Users/hobu/pdal/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.



The screenshot shows a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The command entered is 'C:\>pdal info c:/Users/hobu/pdal/exercises/info/interesting.las ^ More? --query "637301.20, 851217.57, 496.49/3"'. The output is a JSON object representing three nearest points. The first point (index 0) has the following properties:

```
{
  "unnamed": [
    {
      "0": {
        "Blue": 176,
        "Classification": 1,
        "EdgeOfFlightLine": 0,
        "GpsTime": 247185.1549,
        "Green": 158,
        "Intensity": 33,
        "NumberOfReturns": 1,
        "PointSourceId": 7329,
        "Red": 183,
        "ReturnNumber": 1,
        "ScanAngleRank": 10,
        "ScanDirectionFlag": 1,
        "UserData": 124,
        "X": 637379.69,
        "Y": 851215.09,
        "Z": 423.56
      },
      "1": {
        "Blue": 221,
        "Classification": 1,
        "EdgeOfFlightLine": 0,
        "GpsTime": 247185.1549,
        "Green": 158,
        "Intensity": 33,
        "NumberOfReturns": 1,
        "PointSourceId": 7329,
        "Red": 183,
        "ReturnNumber": 1,
        "ScanAngleRank": 10,
        "ScanDirectionFlag": 1,
        "UserData": 124,
        "X": 637379.69,
        "Y": 851215.09,
        "Z": 423.56
      },
      "2": {
        "Blue": 221,
        "Classification": 1,
        "EdgeOfFlightLine": 0,
        "GpsTime": 247185.1549,
        "Green": 158,
        "Intensity": 33,
        "NumberOfReturns": 1,
        "PointSourceId": 7329,
        "Red": 183,
        "ReturnNumber": 1,
        "ScanAngleRank": 10,
        "ScanDirectionFlag": 1,
        "UserData": 124,
        "X": 637379.69,
        "Y": 851215.09,
        "Z": 423.56
      }
    ]
  }
}
```

Notes

1. PDAL uses [JSON](https://en.wikipedia.org/wiki/JSON) (<https://en.wikipedia.org/wiki/JSON>) as the exchange format when printing information from [info](#) (page 27). JSON is a structured, human-readable format that is much simpler than its [XML](https://en.wikipedia.org/wiki/XML) (<https://en.wikipedia.org/wiki/XML>) cousin.
2. The `--query` option of [info](#) (page 27) 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) (<http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html>) data into [LASzip](http://laszip.org) (<http://laszip.org>).

1. Issue the following command in your *OSGeo4W Shell*.

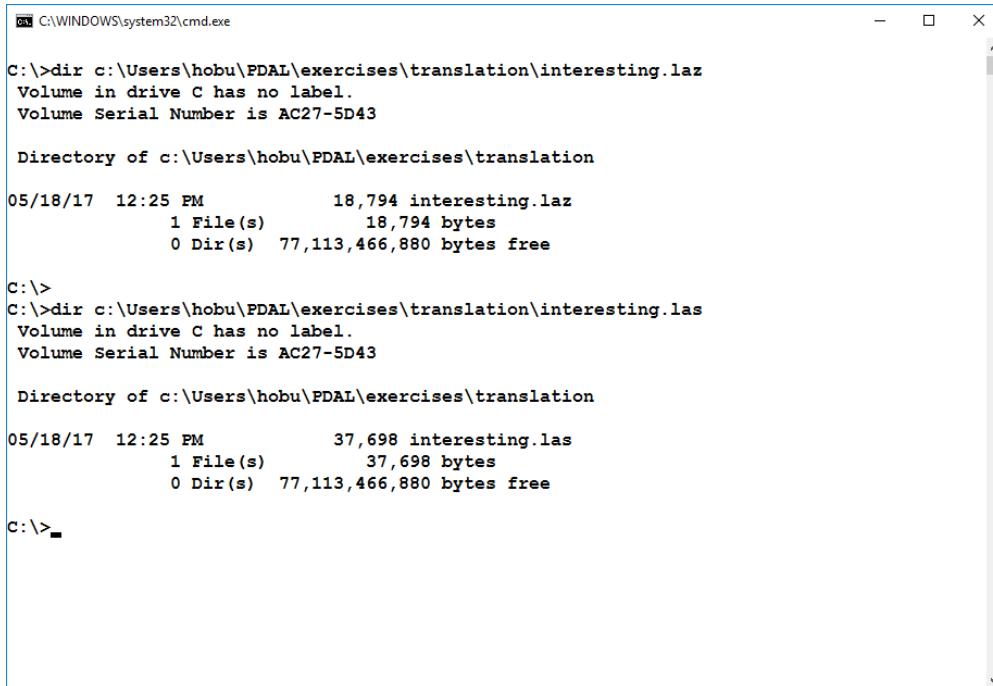
```
pdal translate ^
    c:/Users/hobu/pdal/exercises/translation/interesting.
    ↵las ^
        c:/Users/hobu/pdal/exercises/translation/interesting.
    ↵laz
```

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) (<http://laszip.org>) is an open source solution for compressing this information

2. Verify that the data are in fact compressed:

```
dir c:\Users\hobu\PDAL\exercises\translation\interesting.
    ↵laz

dir c:\Users\hobu\PDAL\exercises\translation\interesting.
    ↵las
```



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command 'dir' is run twice, once for 'interesting.laz' and once for 'interesting.las', both located in the directory 'c:\Users\hobu\PDAL\exercises\translation'. The output shows the file size (18,794 bytes for .laz, 37,698 bytes for .las), the number of files (1), and the amount of free disk space (77,113,466,880 bytes).

```
C:\>dir c:\Users\hobu\PDAL\exercises\translation\interesting.laz
Volume in drive C has no label.
Volume Serial Number is AC27-5D43

Directory of c:\Users\hobu\PDAL\exercises\translation

05/18/17 12:25 PM           18,794 interesting.laz
      1 File(s)           18,794 bytes
      0 Dir(s)  77,113,466,880 bytes free

C:\>
C:\>dir c:\Users\hobu\PDAL\exercises\translation\interesting.las
Volume in drive C has no label.
Volume Serial Number is AC27-5D43

Directory of c:\Users\hobu\PDAL\exercises\translation

05/18/17 12:25 PM           37,698 interesting.las
      1 File(s)           37,698 bytes
      0 Dir(s)  77,113,466,880 bytes free

C:\>
```

See also:

LAS Reading and Writing with PDAL (page 197) 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>

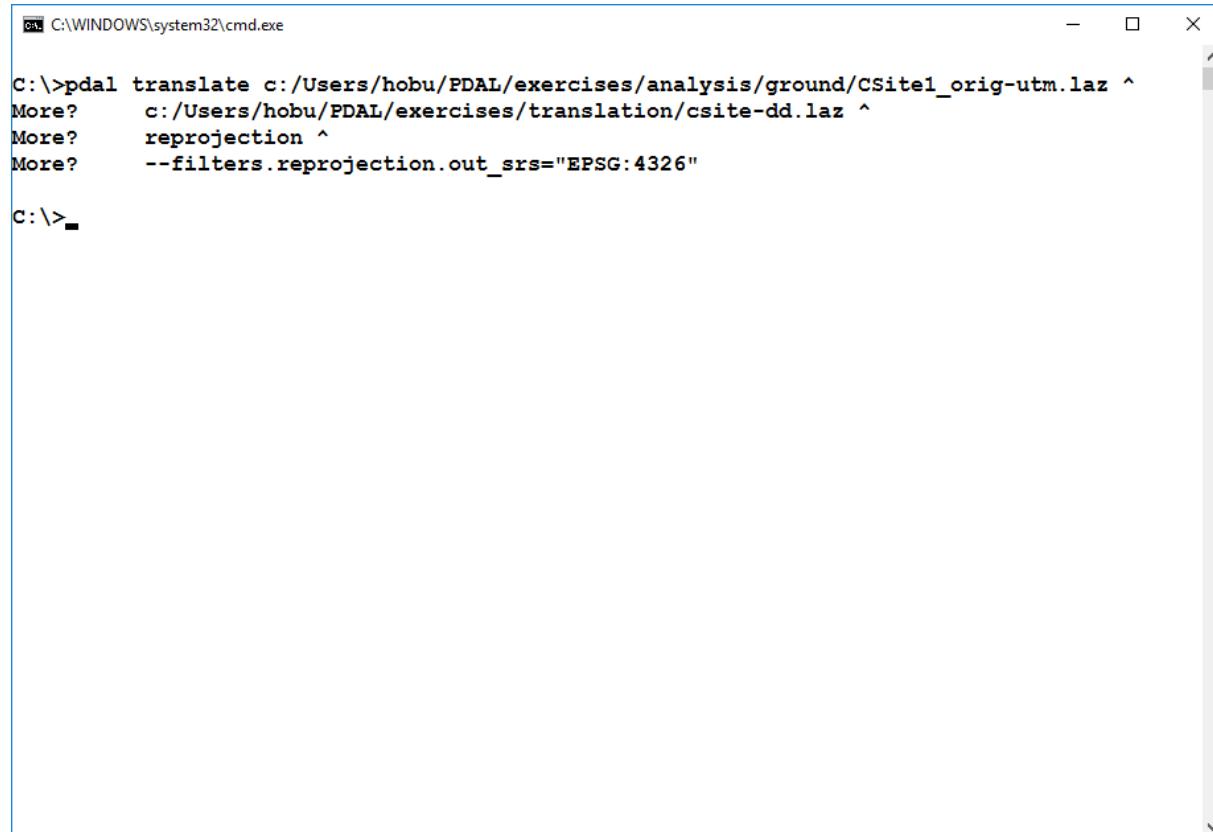
Reprojection

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 *OSGeo4W Shell*.

```
1 pdal translate ^
2   c:/Users/hobu/PDAL/exercises/translation/csite-dd.laz ^
3     reprojection ^
4     --filters.reprojection.out_srs="EPSG:4326"
```



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is:

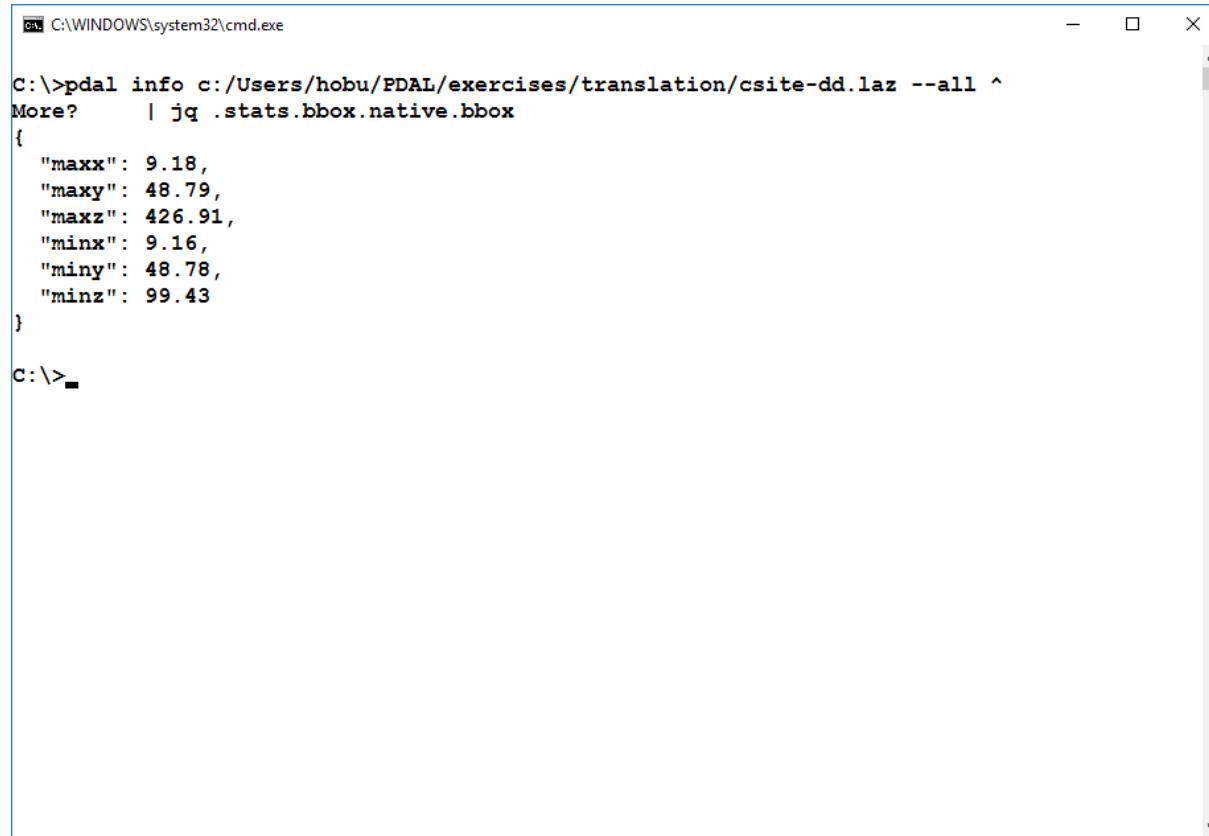
```
C:\>pdal translate c:/Users/hobu/PDAL/exercises/analysis/ground/CSite1_orig-utm.laz ^
More?   c:/Users/hobu/PDAL/exercises/translation/csite-dd.laz ^
More?   reprojection ^
More?   --filters.reprojection.out_srs="EPSG:4326"

C:\>_
```

The command is being processed, with 'More?' prompts appearing for each argument. The window has a standard blue border and a scroll bar on the right.

Unfortunately this doesn't produce the intended results for us. Issue the following `pdal info` command to see why:

```
pdal info c:/Users/hobu/PDAL/exercises/translation/csite-dd.laz --
  ↵all ^
    | jq .stats.bbox.native.bbox
```



The screenshot shows a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The command entered is 'pdal info c:/Users/hobu/PDAL/exercises/translation/csite-dd.laz --all ^ More? | jq .stats.bbox.native.bbox'. The output is a JSON object representing the bounding box of the point cloud:

```
{  
  "maxx": 9.18,  
  "maxy": 48.79,  
  "maxz": 426.91,  
  "minx": 9.16,  
  "miny": 48.78,  
  "minz": 99.43  
}
```

--all dumps all *info* (page 27) 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:

```
C:\>pdal info c:/Users/hobu/PDAL/exercises/translation/csite-dd.laz -p 0
{
  "filename": "c:/Users/hobu/PDAL/exercises/translation/csite-dd.laz",
  "pdal_version": "1.5.0 (git-version: 15fe7d)",
  "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
    }
  }
}

C:\>
```

Some formats, like *writers.las* (page 85) 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.

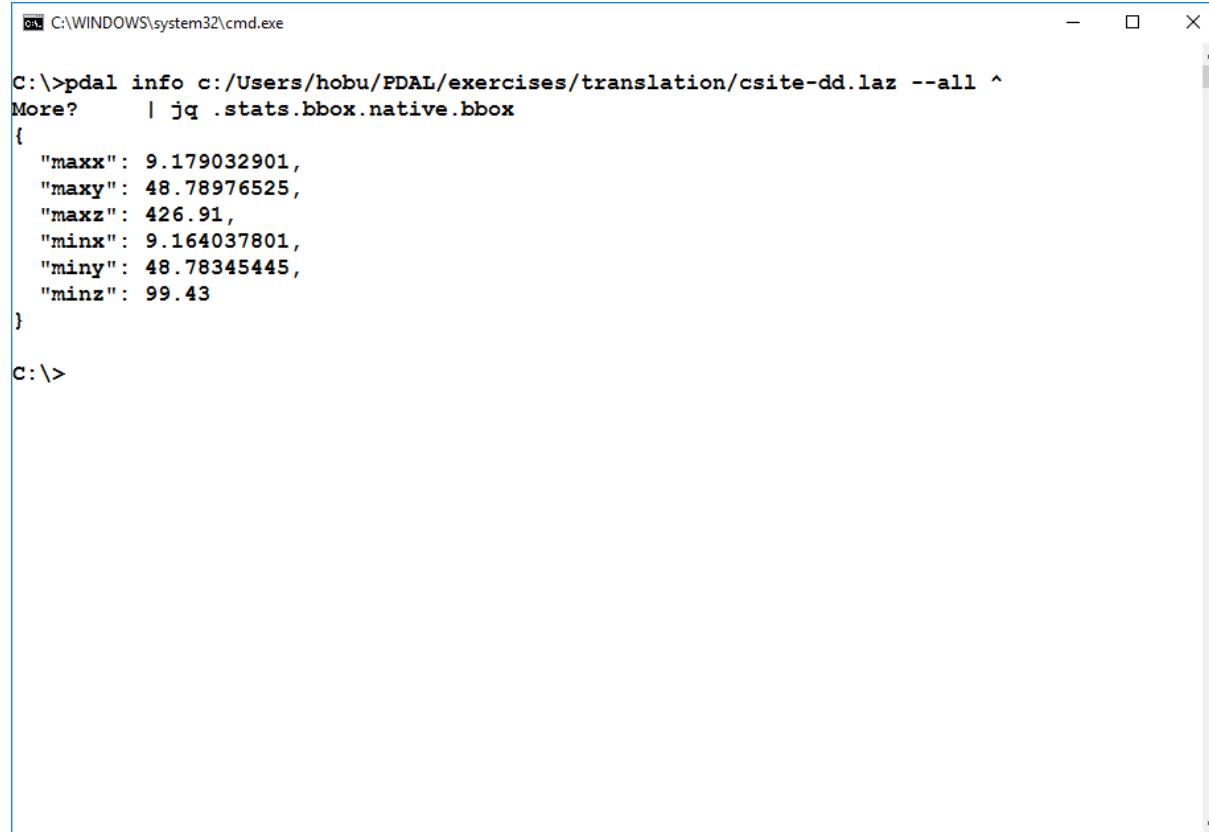
```
1 pdal translate ^
2   c:/Users/hobu/PDAL/exercises/analysis/ground/CSite1_orig-utm.laz ^
3   ↵^
4   c:/Users/hobu/PDAL/exercises/translation/csite-dd.laz ^
5   reprojection ^
6   --filters.reprojection.out_srs="EPSG:4326" ^
7   --writers.las.scale_x=0.0000001 ^
8   --writers.las.scale_y=0.0000001 ^
9   --writers.las.offset_x="auto" ^
10  --writers.las.offset_y="auto"
```



```
C:\>pdal translate ^
More?      c:/Users/hobu/PDAL/exercises/analysis/ground/CSite1_orig-utm.laz ^
More?      c:/Users/hobu/PDAL/exercises/translation/csite-dd.laz ^
More?      reprojection ^
More?      --filters.reprojection.out_srs="EPSG:4326" ^
More?      --writers.las.scale_x=0.0000001 ^
More?      --writers.las.scale_y=0.0000001 ^
More?      --writers.las.offset_x="auto" ^
More?      --writers.las.offset_y="auto"

C:\>
```

Run the *pdal info* command again to verify the X, Y, and Z dimensions:



```
C:\>pdal info c:/Users/hobu/PDAL/exercises/translation/csite-dd.laz --all ^
More?      | jq .stats.bbox.native.bbox
{
  "maxx": 9.179032901,
  "maxy": 48.78976525,
  "maxz": 426.91,
  "minx": 9.164037801,
  "miny": 48.78345445,
  "minz": 99.43
}

C:\>
```

Notes

1. *filters.reprojection* (page 173) 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) (<http://proj4.org>) library for reprojection. This library includes the capability to do both vertical and horizontal datum transformations.

Greyhound

Exercise

This exercise uses PDAL to fetch data from a Greyhound server. Greyhound is a web server for point cloud data. You can learn more about what it is by visiting
<http://lidarnews.com/articles/open-source-point-cloud-web-services-with-greyhound/>

See the Dublin data used in this example in your browser at

<http://potree.entwine.io/data/dublin.html>

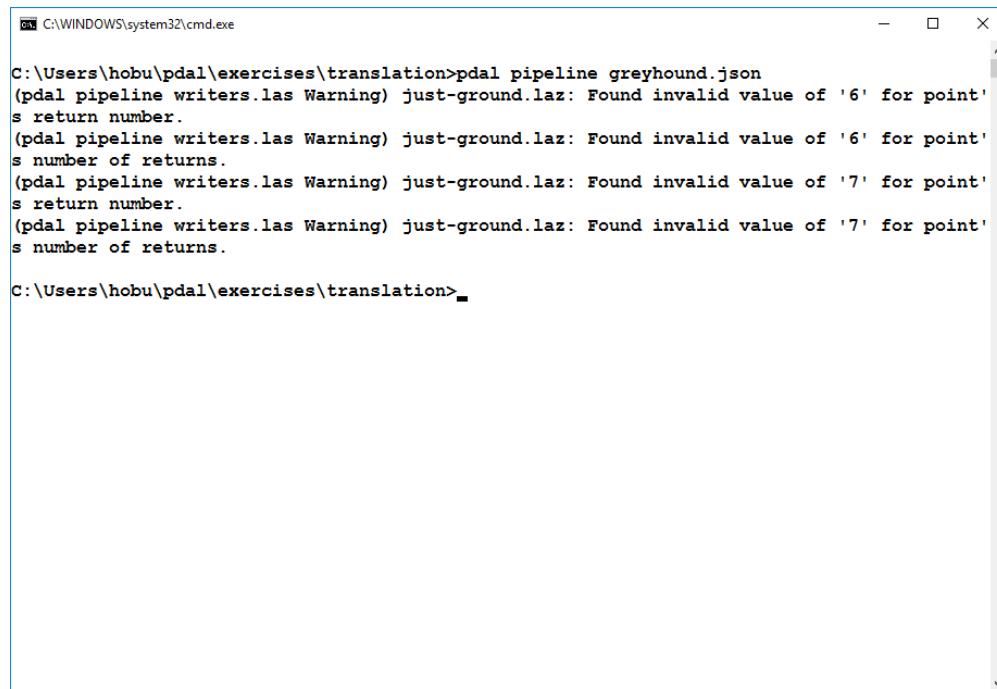
1. View the `greyhound.json` file in your editor

```
{
    "pipeline": [
        {
            "type": "readers.greyhound",
            "url": "data.greyhound.io",
            "depth_begin": 0,
            "depth_end": 11,
            "resource": "dublin",
            "filter": {
                "Classification": 2
            }
        },
        {
            "type": "writers.las",
            "compression": "true",
            "minor_version": "2",
            "dataformat_id": "0",
            "filename": "just-ground.laz"
        }
    ]
}
```

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 Greyhound server at <http://data.greyhound.io>

2. Issue the following command in your *OSGeo4W Shell*.

```
pdal pipeline greyhound.json
```



A screenshot of a Windows command-line interface (cmd.exe) window. The window title is "C:\WINDOWS\system32\cmd.exe". The command entered is "pdal pipeline greyhound.json". The output shows several warnings from the pdal pipeline writer, specifically regarding invalid values for point return numbers (6 and 7). The command prompt ends with a greater-than sign and a underscore.

```
C:\Users\hobu\pdal\exercises\translation>pdal pipeline greyhound.json
(pdal pipeline writers.las Warning) just-ground.laz: Found invalid value of '6' for point's return number.
(pdal pipeline writers.las Warning) just-ground.laz: Found invalid value of '6' for point's number of returns.
(pdal pipeline writers.las Warning) just-ground.laz: Found invalid value of '7' for point's return number.
(pdal pipeline writers.las Warning) just-ground.laz: Found invalid value of '7' for point's number of returns.

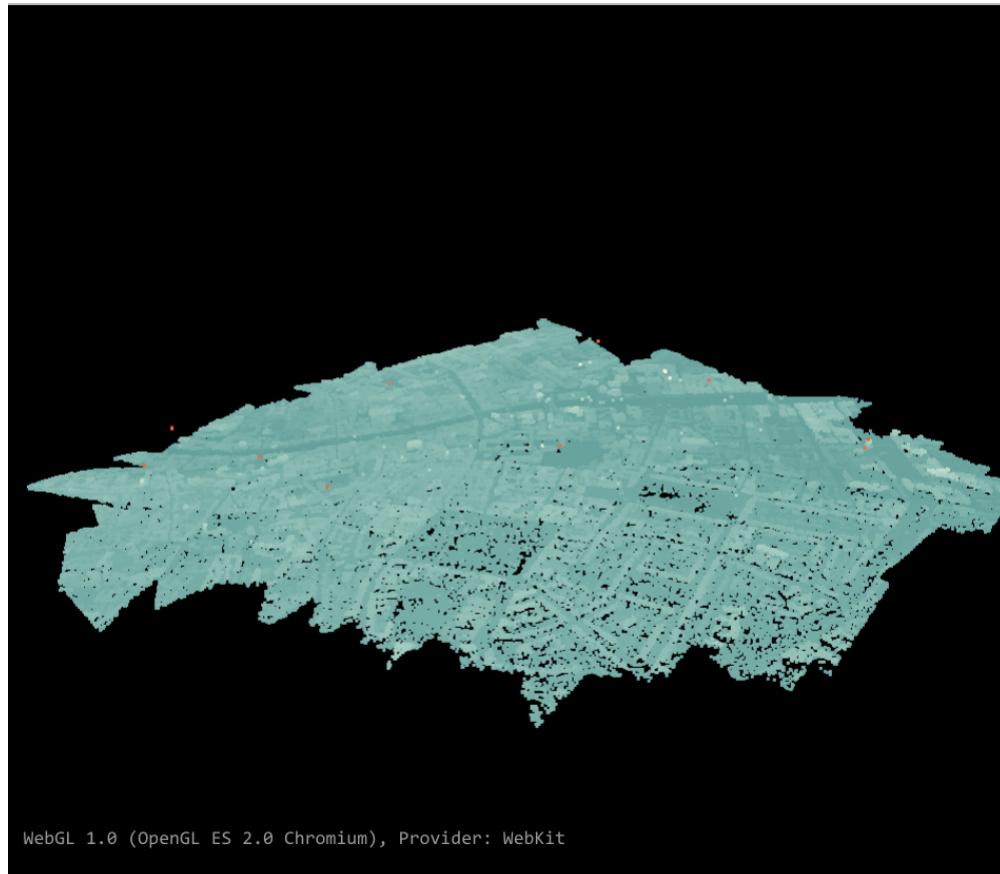
C:\Users\hobu\pdal\exercises\translation>_
```

3. Verify that the data look ok:

```
pdal info just-ground.las | ^
    jq .stats.bbox.native
```

```
PS Select C:\WINDOWS\system32\cmd.exe
C:\Users\hobu\pdal\exercises\translation>pdal info just-ground.laz | jq .stats.bbox.native
(pdal info readers.las Warning) just-ground.laz: Found invalid value of '6' for point's return
number.
(pdal info readers.las Warning) just-ground.laz: Found invalid value of '6' for point's number
of returns.
(pdal info readers.las Warning) just-ground.laz: Found invalid value of '7' for point's return
number.
(pdal info readers.las Warning) just-ground.laz: Found invalid value of '7' for point's number
of returns.
{
  "bbox": {
    "maxx": -693813.25,
    "maxy": 7049930.61,
    "maxz": 358.34,
    "minx": -699477.8,
    "miny": 7044490.98,
    "minz": -96.88
  },
  "boundary": {
    "coordinates": [
      [
        [
          [
            [
              [
                [
                  [
                    [
                      [
                        [
                          [
                            [
                              [
                                [
                                  [
                                    [
                                      [
                                        [
                                          [
                                            [
                                              [
                                                [
                                                  [
                                                    [
                                                      [
                                                        [
                                                          [
                                                            [
                                                              [
                                                                [
                                                                  [
                                                                    [
                                                                      [
                                                                        [
                                                                          [
                                                                            [
                                                                              [
                                                                                [
                                                                                  [
                                                                                    [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
                                                                                      [
................................................................
```

4. Visualize the data in <http://plas.io>



WebGL 1.0 (OpenGL ES 2.0 Chromium), Provider: WebKit

Notes

1. *readers.greyhound* (page 56) contains more detailed documentation about how to use PDAL's [Greyhound](http://greyhound.io/) (<http://greyhound.io/>) reader .
2. As `depth_end` gets larger, the number of possible points goes up by nearly a factor of 4. Use the `bounds` option of the reader to split up the boxes you are querying to decrease the potential number of points a query might return.

Analysis

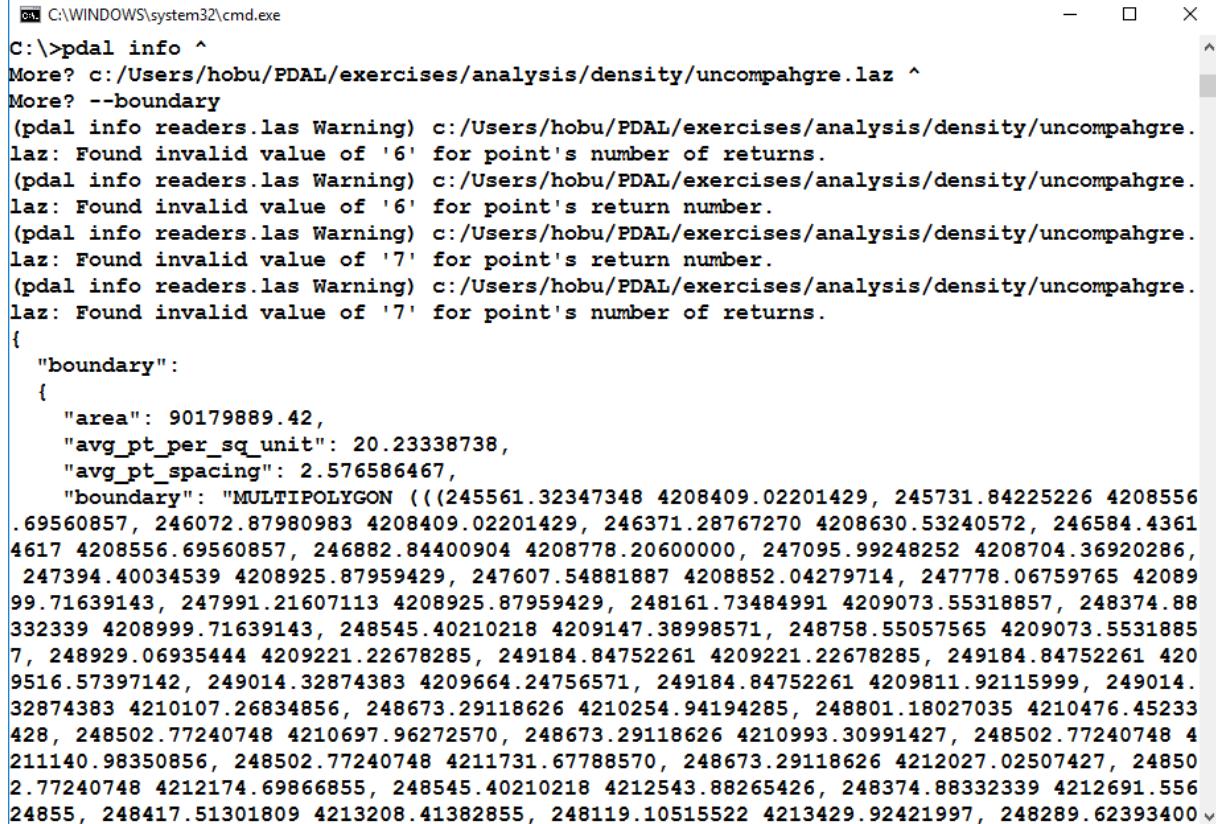
Finding the boundary

This exercise uses PDAL to find a tight-fighting 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.

```
1 pdal info ^
2     c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz ^
3     --boundary
```

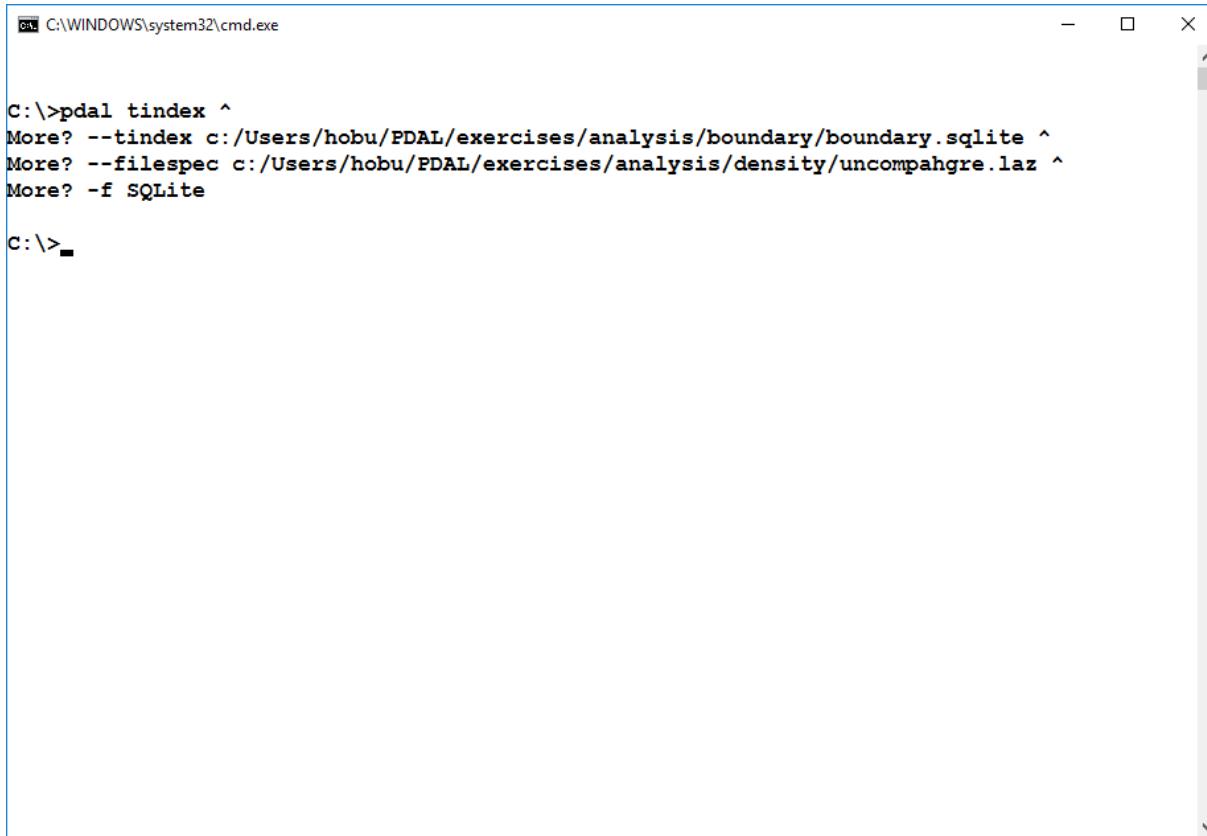


```
C:\>pdal info ^
More? c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz ^
More? --boundary
(pdal info readers.las Warning) c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz: Found invalid value of '6' for point's number of returns.
(pdal info readers.las Warning) c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz: Found invalid value of '6' for point's return number.
(pdal info readers.las Warning) c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz: Found invalid value of '7' for point's return number.
(pdal info readers.las Warning) c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz: Found invalid value of '7' for point's number of returns.
{
  "boundary": {
    "area": 90179889.42,
    "avg_pt_per_sq_unit": 20.23338738,
    "avg_pt_spacing": 2.576586467,
    "boundary": "MULTIPOLYGON (((245561.32347348 4208409.02201429, 245731.84225226 4208556.69560857, 246072.87980983 4208409.02201429, 246371.28767270 4208630.53240572, 246584.43614617 4208556.69560857, 246882.84400904 4208778.20600000, 247095.99248252 4208704.36920286, 247394.40034539 4208925.87959429, 247607.54881887 4208852.04279714, 247778.06759765 420899.71639143, 247991.21607113 4208925.87959429, 248161.73484991 4209073.55318857, 248374.88332339 4208999.71639143, 248545.40210218 4209147.38998571, 248758.55057565 4209073.55318857, 248929.06935444 4209221.22678285, 249184.84752261 4209221.22678285, 249184.84752261 4209516.57397142, 249014.32874383 4209664.24756571, 249184.84752261 4209811.92115999, 249014.32874383 4210107.26834856, 248673.29118626 4210254.94194285, 248801.18027035 4210476.45233428, 248502.77240748 4210697.96272570, 248673.29118626 4210993.30991427, 248502.77240748 4211140.98350856, 248502.77240748 4211731.67788570, 248673.29118626 4212027.02507427, 248502.77240748 4212174.69866855, 248545.40210218 4212543.88265426, 248374.88332339 4212691.55624855, 248417.51301809 4213208.41382855, 248119.10515522 4213429.92421997, 248289.62393400 v
```

... 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 134). We can leverage this capability to output a contiguous boundary of the uncompahgre.laz file.

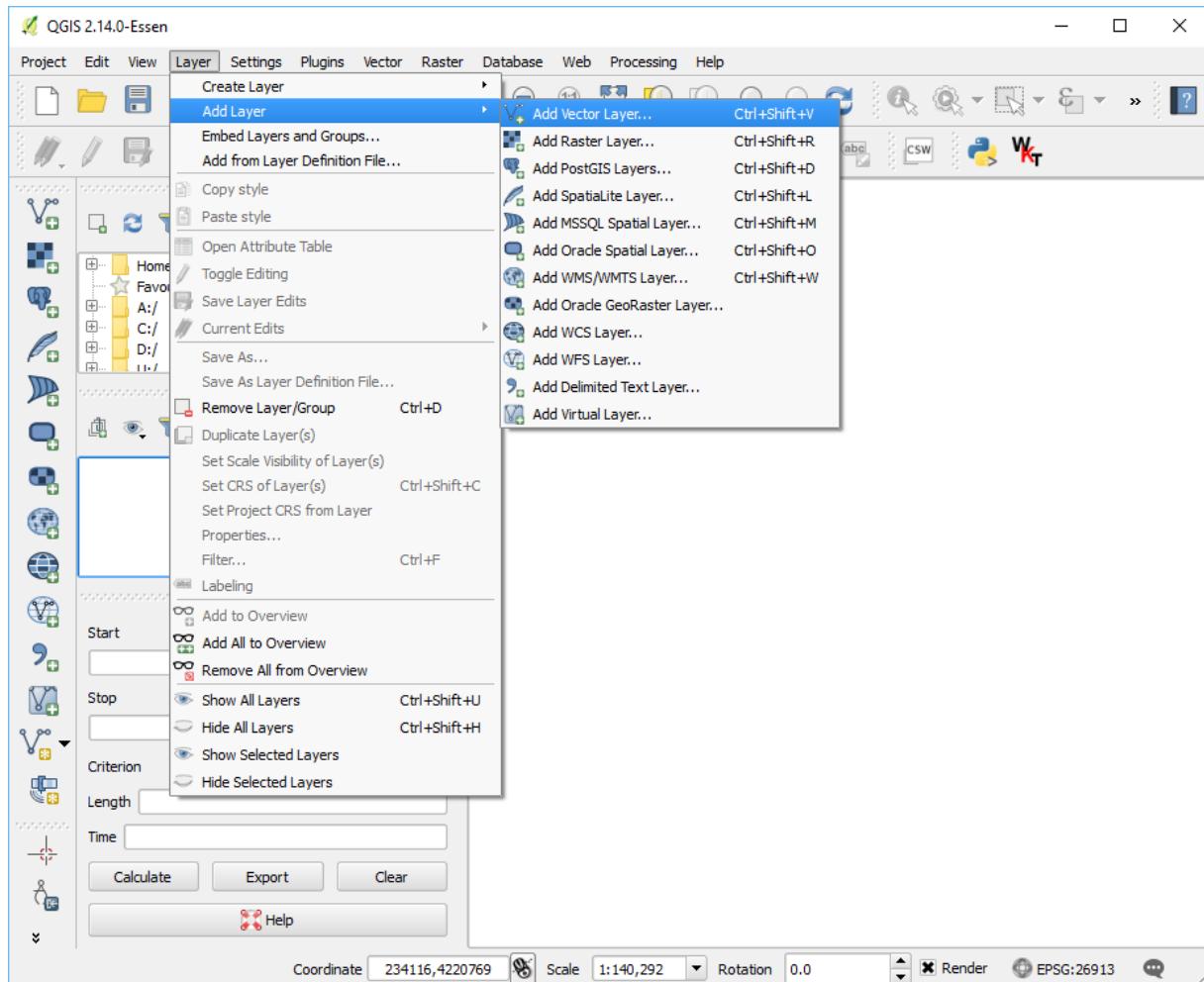
```
1 pdal tindex ^
2   --tindex c:/Users/hobu/PDAL/exercises/analysis/boundary/boundary.
3   ↵sqlite ^
4   --filespec c:/Users/hobu/PDAL/exercises/analysis/density/
5   ↵uncompahgre.laz ^
6   -f SQLite
```



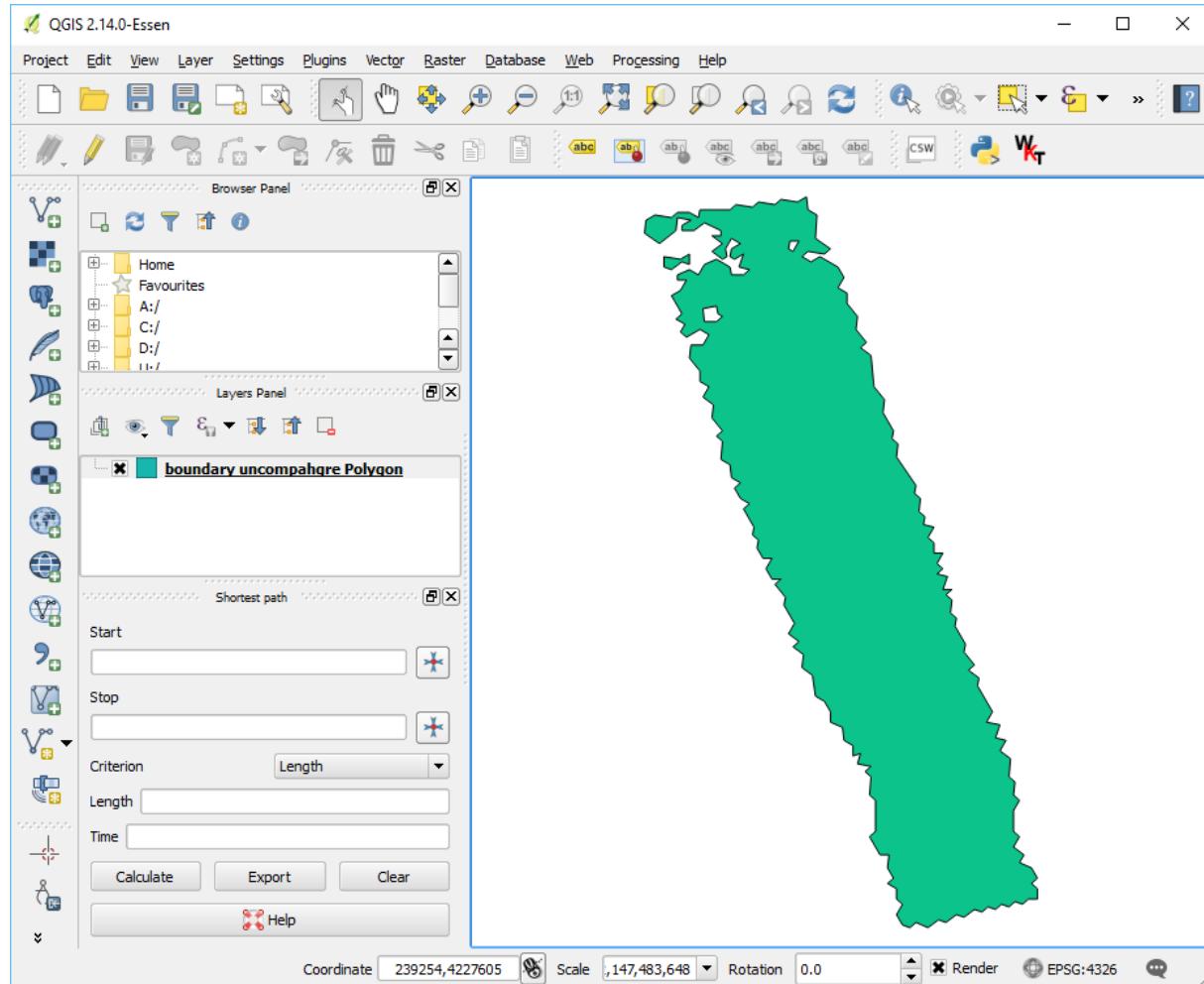
The screenshot shows a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The command entered is 'pdal tindex ^ More? --tindex c:/Users/hobu/PDAL/exercises/analysis/boundary/boundary.sqlite ^ More? --filespec c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz ^ More? -f SQLite'. The prompt 'C:\>' is visible at the bottom.

Once we've run the *tindex* (page 34), we can now visualize our output:

Open qgis and select *Add Vector Layer*:



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 134) can also be used by the `density` (page 25) to generate a tessellated surface. See the *Visualizing acquisition density* (page 282) example for steps to achieve this.
3. The `tindex` (page 34) 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).

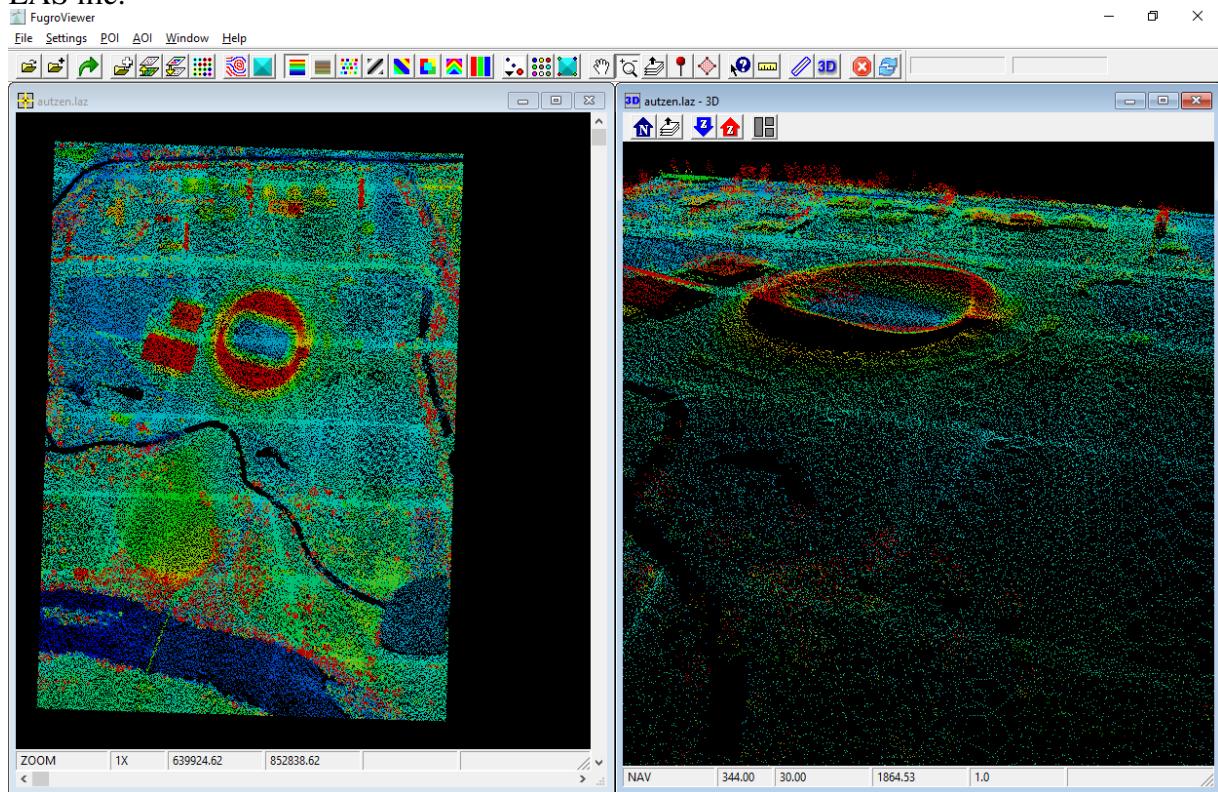
Clipping data with polygons

This exercise uses PDAL to apply to clip data with polygon geometries.

Note: This exercise is an adaption of the [PDAL tutorial](#) (page 223).

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.



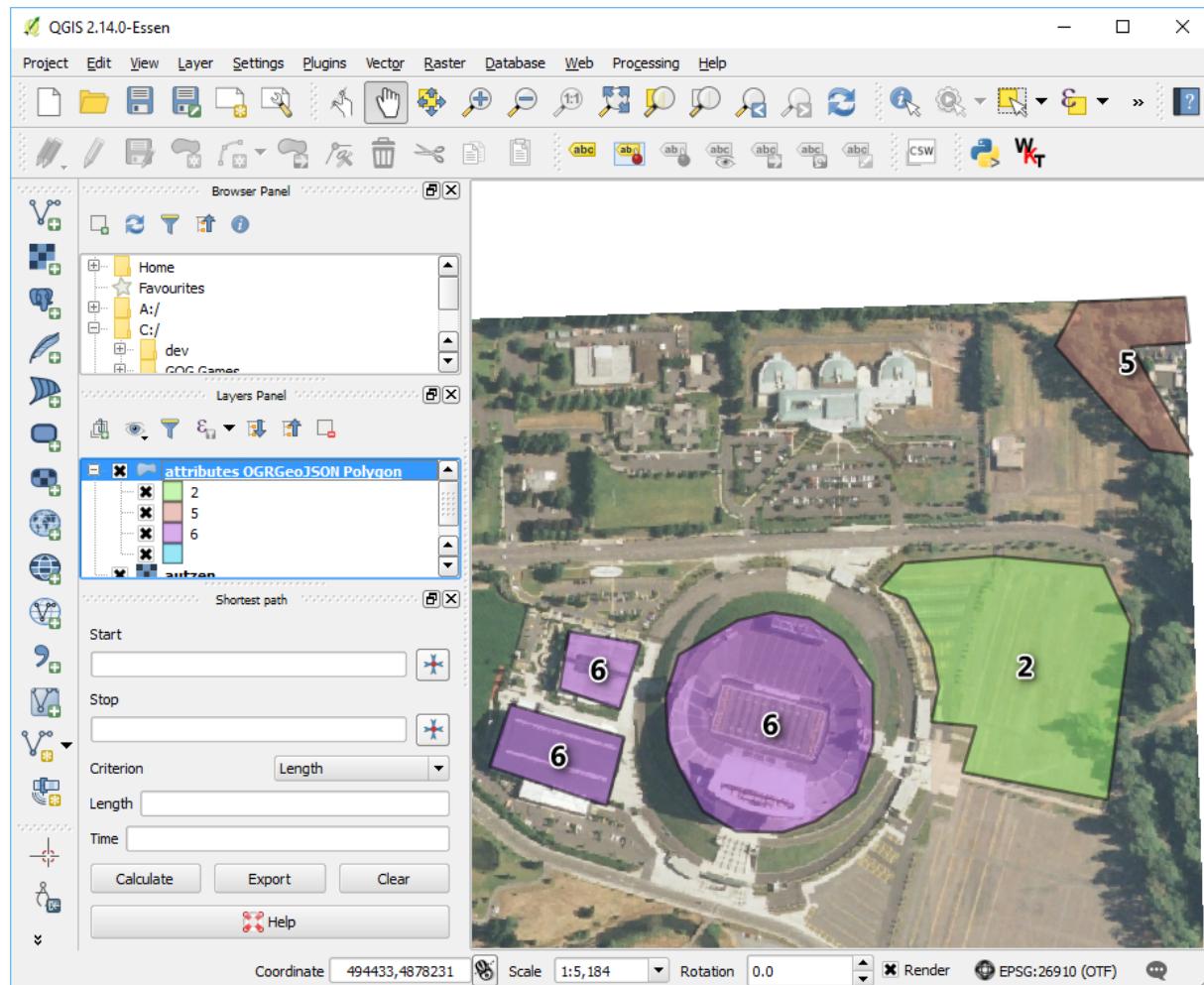
Data preparation

The data are mixed in two different coordinate systems. The [LAZ](#) (page 59) 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:

```
<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=399999.999999999 +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.json` file. 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

```
{  
    "pipeline": [  
        "c:/Users/hobu/PDAL/exercises/analysis/clipping/autzen.laz",  
        {  
            "column": "CLS",  
            "datasource": "c:/Users/hobu/PDAL/exercises/analysis/  
clipping/attributes.vrt",  
            "dimension": "Classification",  
            "layer": "OGRGeoJSON",  
            "type": "filters.overlay"  
        },  
        {  
            "limits": "Classification[6:6]",  
            "type": "filters.range"  
        },  
        "c:/Users/hobu/PDAL/exercises/analysis/clipping/stadium.las"  
    ]  
}
```

Note: This pipeline is available in your workshop materials in the [./exercises/analysis/clipping/clipping.json](#) file.

1. Reader

`autzen.laz` is the [LASzip](#) (<http://laszip.org>) file we will clip.

2. `filters.overlay`

The [`filters.overlay`](#) (page 154) filter allows you to assign values for coincident polygons. Using the VRT we defined in [Data preparation](#) (page 269), [`filters.overlay`](#) (page 154) 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 170) to keep points with Classification values in the range of 6 : 6.

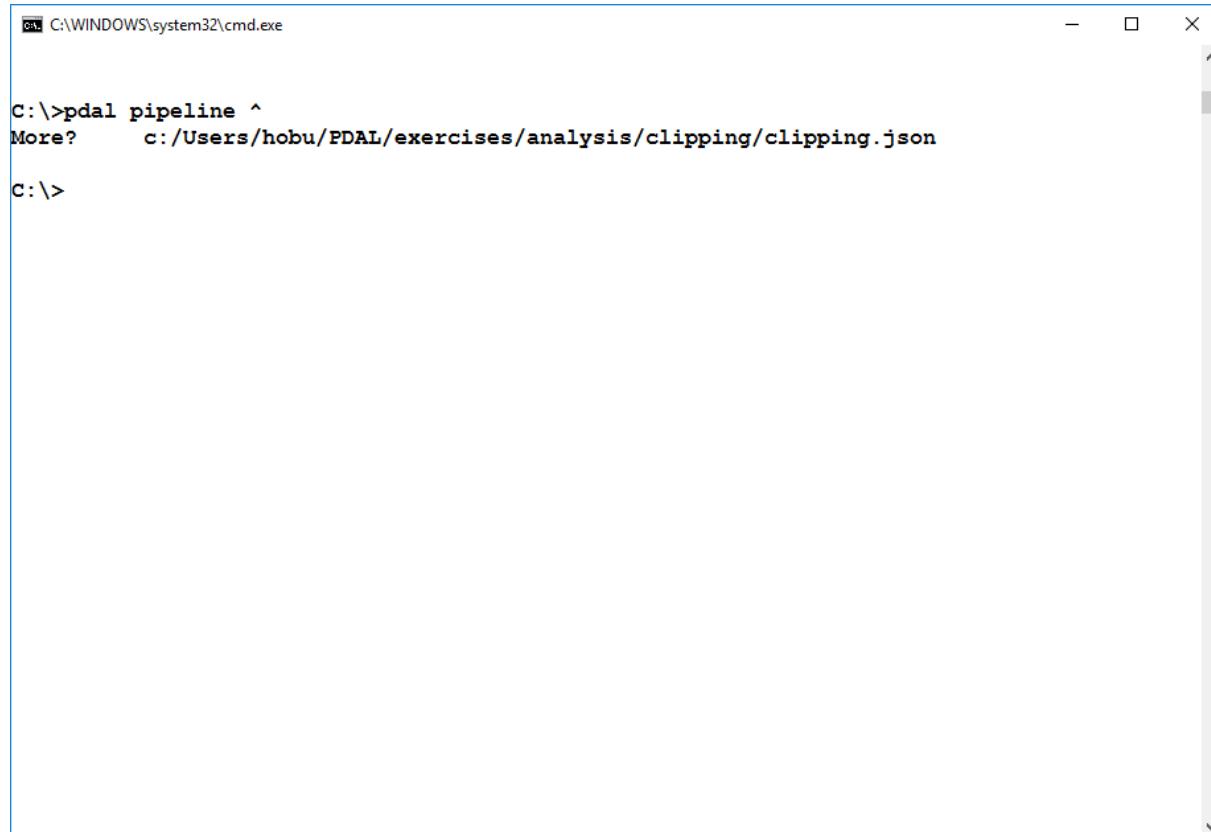
3. Writer

- We will write our content back out using an [writers.las](#) (page 85).

Execution

Invoke the following command, substituting accordingly, in your *OSGeo4W Shell*:

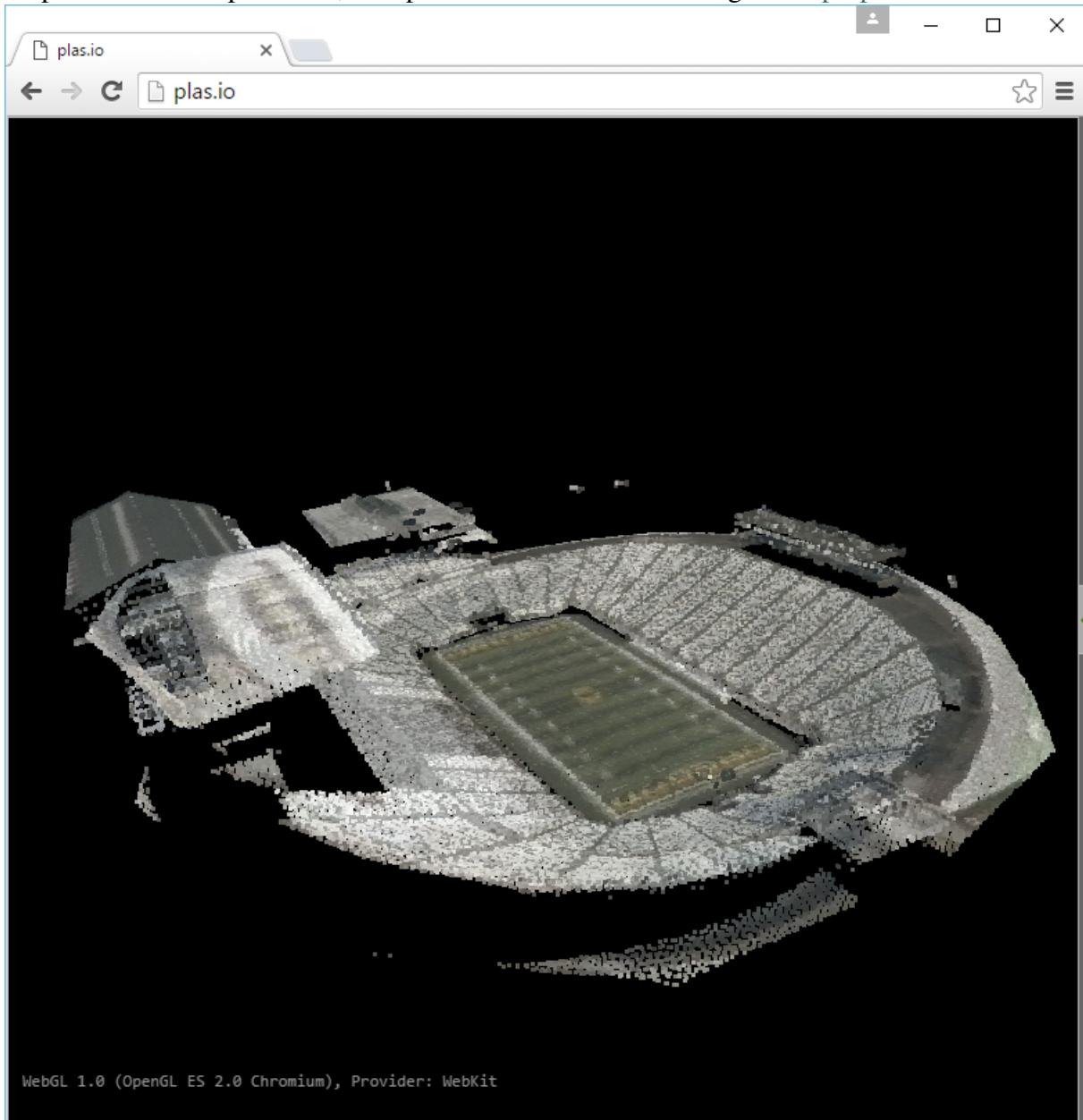
```
1 pdal pipeline ^
2     c:/Users/hobu/PDAL/exercises/analysis/clipping/clipping.json
```



```
C:\>pdal pipeline ^
More?      c:/Users/hobu/PDAL/exercises/analysis/clipping/clipping.json
C:\>
```

Visualization

Use one of the point cloud visualization tools you installed to take a look at your `c:/Users/Howard/PDAL/exercises/analysis/clipping/stadium.las` output. In the example below, we opened the file to view it using the <http://plas.io> website.



Notes

1. `filters.overlay` (page 154) 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 104) 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.

```
1  {
2      "pipeline": [
3          "c:/Users/hobu/PDAL/exercises/analysis/colorization/
4              ↳uncompahgre.laz",
5              {
6                  "type": "filters.colorization",
7                  "raster": "c:/Users/hobu/PDAL/exercises/analysis/
8                      ↳colorization/casi-2015-04-29-weekly-mosaic.tif"
9              },
10             {
11                 "type": "filters.range",
12                 "limits": "Red[1:]"
13             },
14             {
15                 "type": "writers.las",
16                 "compression": "true",
17                 "minor_version": "2",
18                 "dataformat_id": "3",
19                 "filename": "c:/Users/hobu/PDAL/exercises/analysis/
20                     ↳colorization/uncompahgre-colored.laz"
21             }
22         ]
23     }
```

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.

```
"c:/Users/hobu/PDAL/exercises/analysis/colorization/uncompahgre.laz",
```

2. filters.colorization

The [filters.colorization](#) (page 113) 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": "c:/Users/hobu/PDAL/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 170) 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",
```

```
    "filename": "c:/Users/hobu/PDAL/exercises/colorization/analysis/  
    ↵uncompahgre-colored.laz"  
}
```

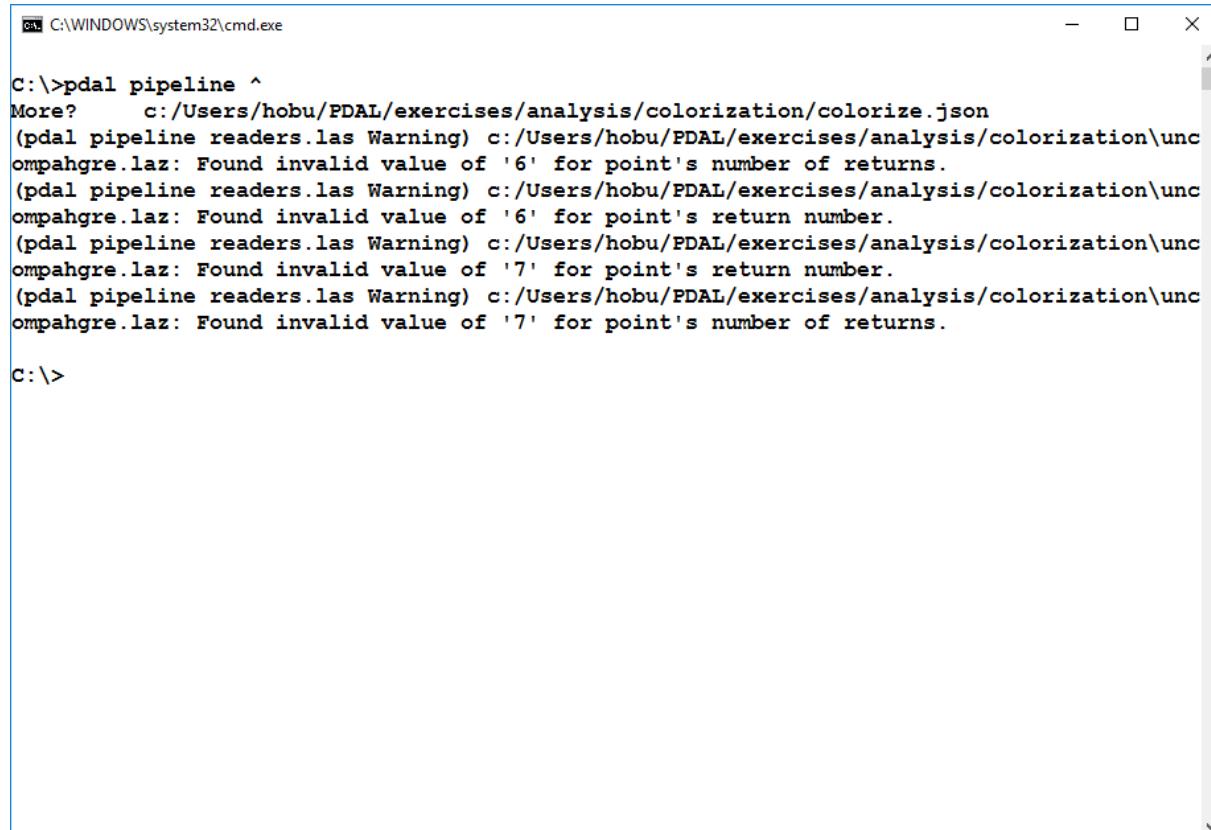
1. compression: [LASzip](http://laszip.org) (<http://laszip.org>) data is ~6x smaller than ASPRS LAS.
2. minor_version: We want to make sure to output LAS 1.2, which will provide the widest compatibility with other softwares that can consume LAS.
3. dataformat_id: Format 3 supports both time and color information

Note: [writers.las](#) (page 85) provides a number of possible options to control how your LAS files are written.

Execution

Invoke the following command, substituting accordingly, in your *Docker Quickstart Terminal*:

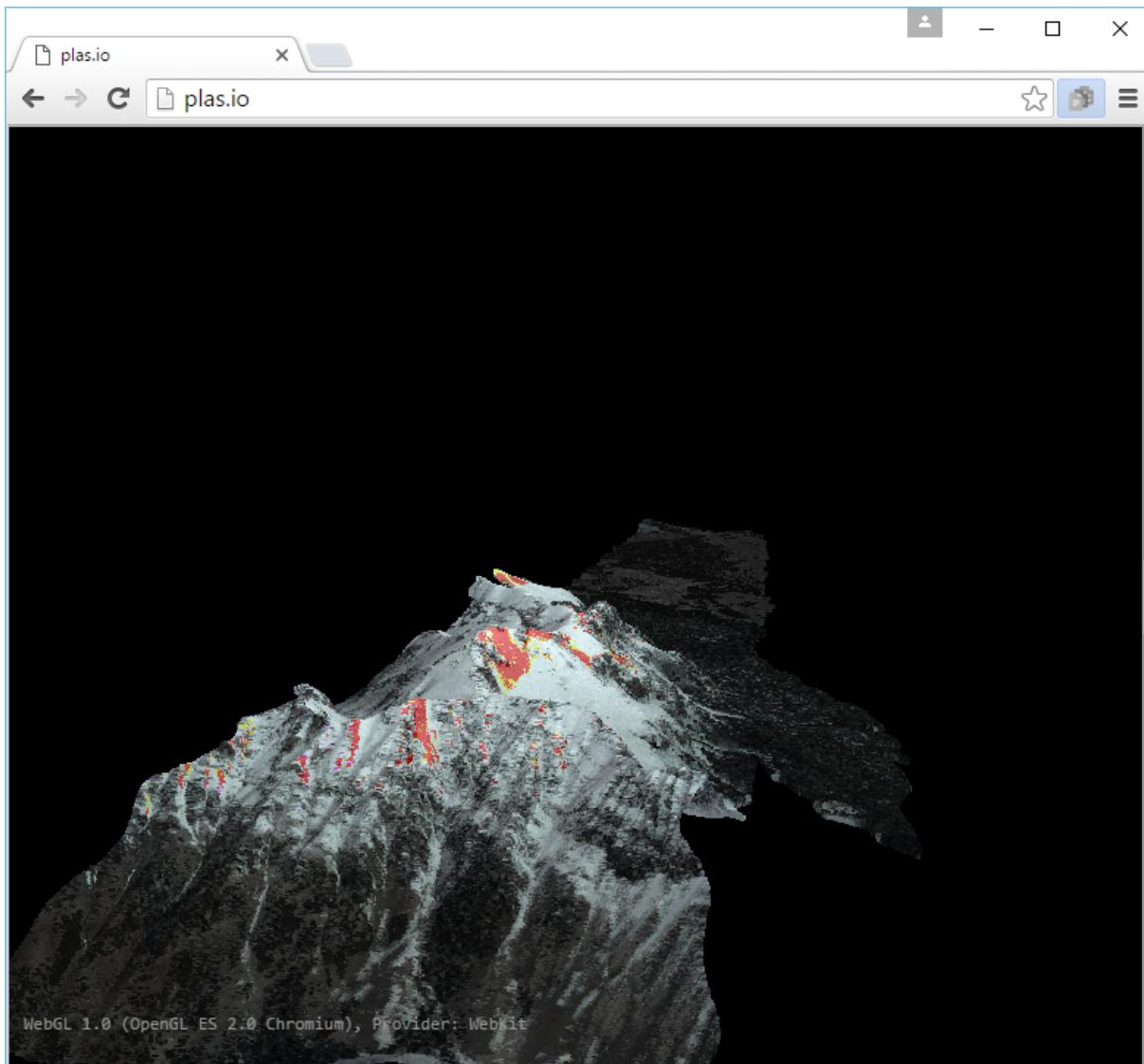
```
1 pdal pipeline ^  
2     c:/Users/hobu/PDAL/exercises/analysis/colorization/colorize.json
```



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is 'pdal pipeline ^ c:/Users/hobu/PDAL/exercises/analysis/colorization/colorize.json'. The output shows several warning messages from PDAL's readers.las component regarding invalid values for point's number of returns and return number, specifically mentioning values 6 and 7. The window has standard minimize, maximize, and close buttons at the top right.

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 113) for more information.

4. These data are of [Uncompahgre Basin](#) (https://en.wikipedia.org/wiki/Uncompahgre_River) courtesy of the [NASA Airborne Snow Observatory](#) (<http://aso.jpl.nasa.gov/>).

Removing noise

This exercise uses PDAL to remove unwanted noise in an airborne LiDAR collection.

Exercise

PDAL provides the [outlier filter](#) (page 151) to apply a statistical filter to data.

Because this operation is somewhat complex, we are going to use a pipeline to define it.

```
{
  "pipeline": [
    "c:/Users/hobu/PDAL/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": "c:/Users/hobu/PDAL/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.

```
"c:/Users/hobu/PDAL/exercises/analysis/denoising/18TWK820985.laz",
```

2. filters.outlier

The PDAL *outlier filter* (page 151) 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 170) can remove these noise points by constructing a *range* (page 172) with the value Classification![7:7], which passes every point with a Classification value **not** equal to 7.

Even with the *filters.outlier* (page 151) 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 172) to keep only points that are within the range $-100 \leq Z \leq 3000$.

Both *ranges* (page 172) are passed as a comma-separated list to the *range filter* (page 170) 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:

```
{  
    "type": "writers.las",  
    "compression": "true",  
    "minor_version": "2",  
    "dataformat_id": "0",  
    "filename": "c:/Users/hobu/PDAL/exercises/analysis/denoising/  
    ↪clean.laz"  
}
```

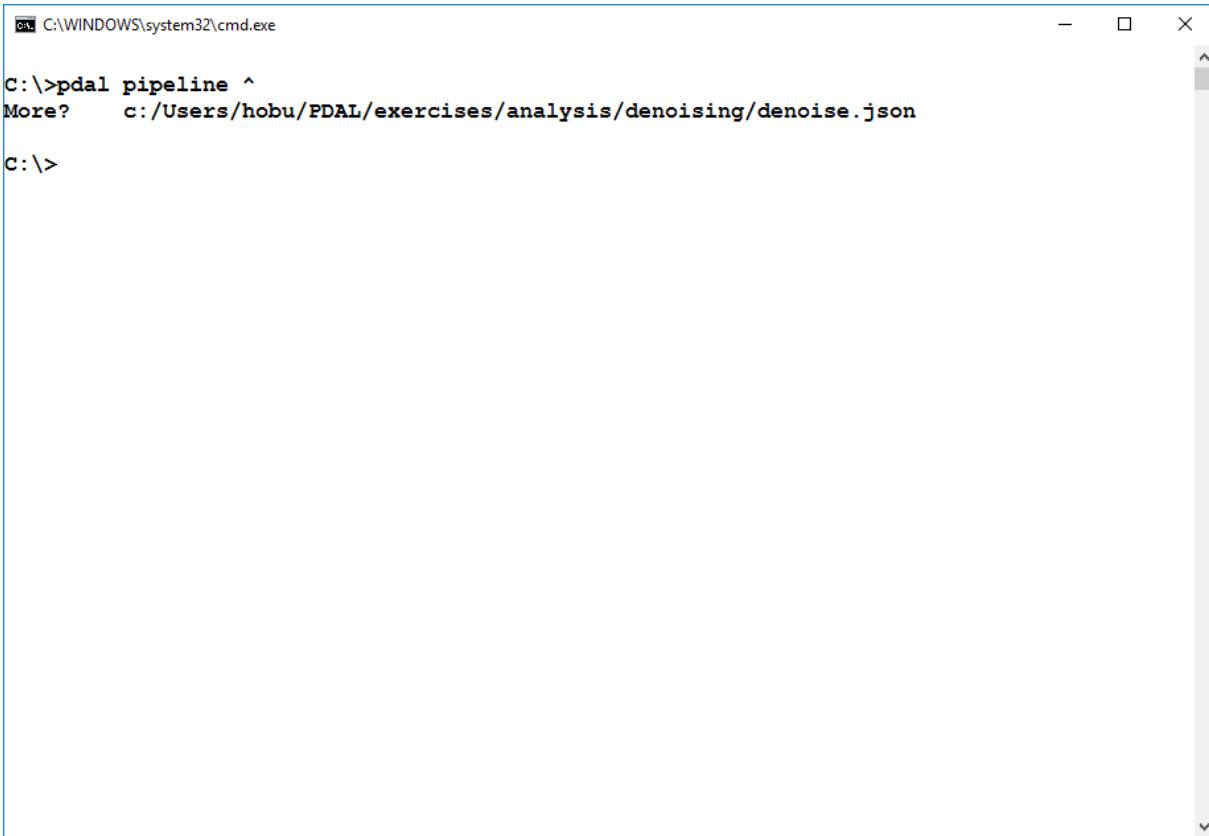
1. `compression`: [LASzip](http://laszip.org) (<http://laszip.org>) data is ~6x smaller than ASPRS LAS.
2. `minor_version`: We want to make sure to output LAS 1.2, which will provide the widest compatibility with other softwares that can consume LAS.
3. `dataformat_id`: Format 3 supports both time and color information

Note: [writers.las](#) (page 85) provides a number of possible options to control how your LAS files are written.

Execution

Invoke the following command, substituting accordingly, in your *OSGeo4W Shell*:

```
pdal pipeline ^  
c:/Users/hobu/PDAL/exercises/analysis/denoising/denoise.json
```

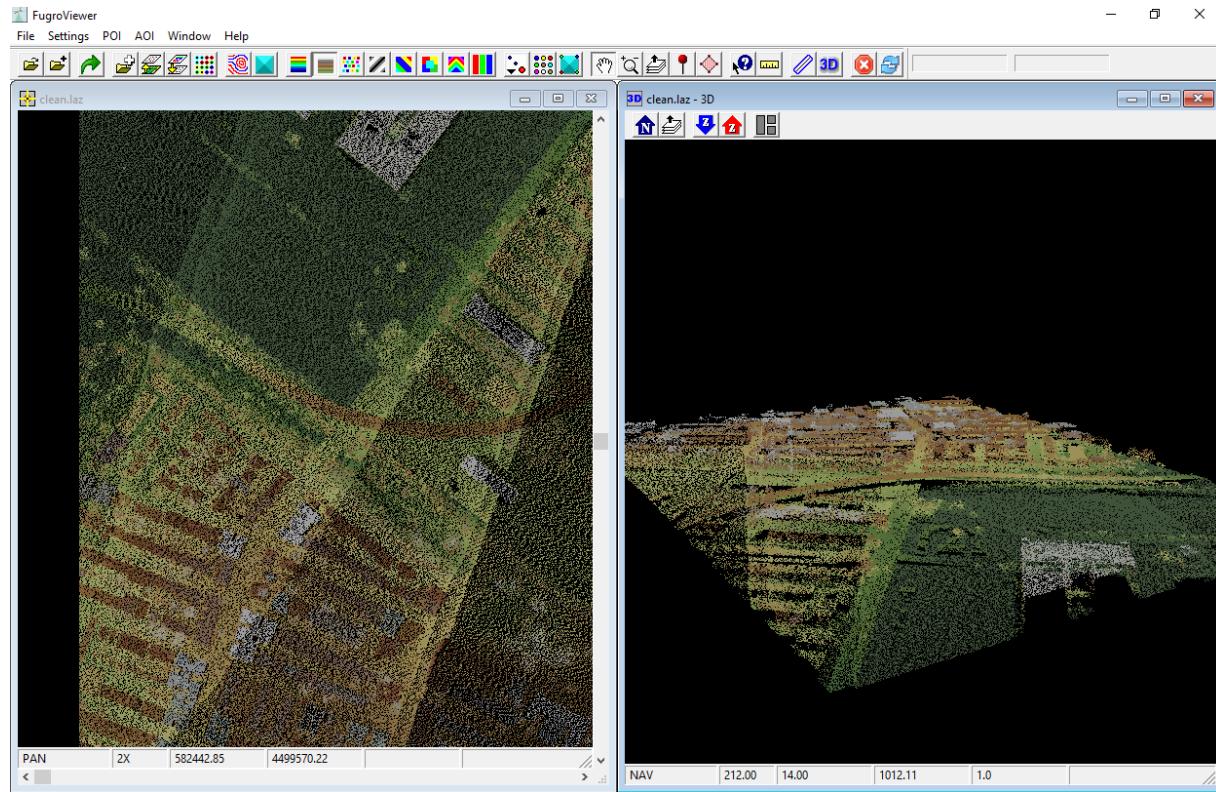


A screenshot of a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text:

```
C:\>pdal pipeline ^
More?  c:/Users/hobu/PDAL/exercises/analysis/denoising/denoise.json
c:\>
```

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/) (<http://www.fugroviewer.com/>)



Notes

1. Control the aggressiveness of the algorithm with the `mean_k` parameter.
2. `filters.outlier` (page 151) requires the entire set in memory to process. If you have really large files, you are going to need to `split` (page 178) 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.

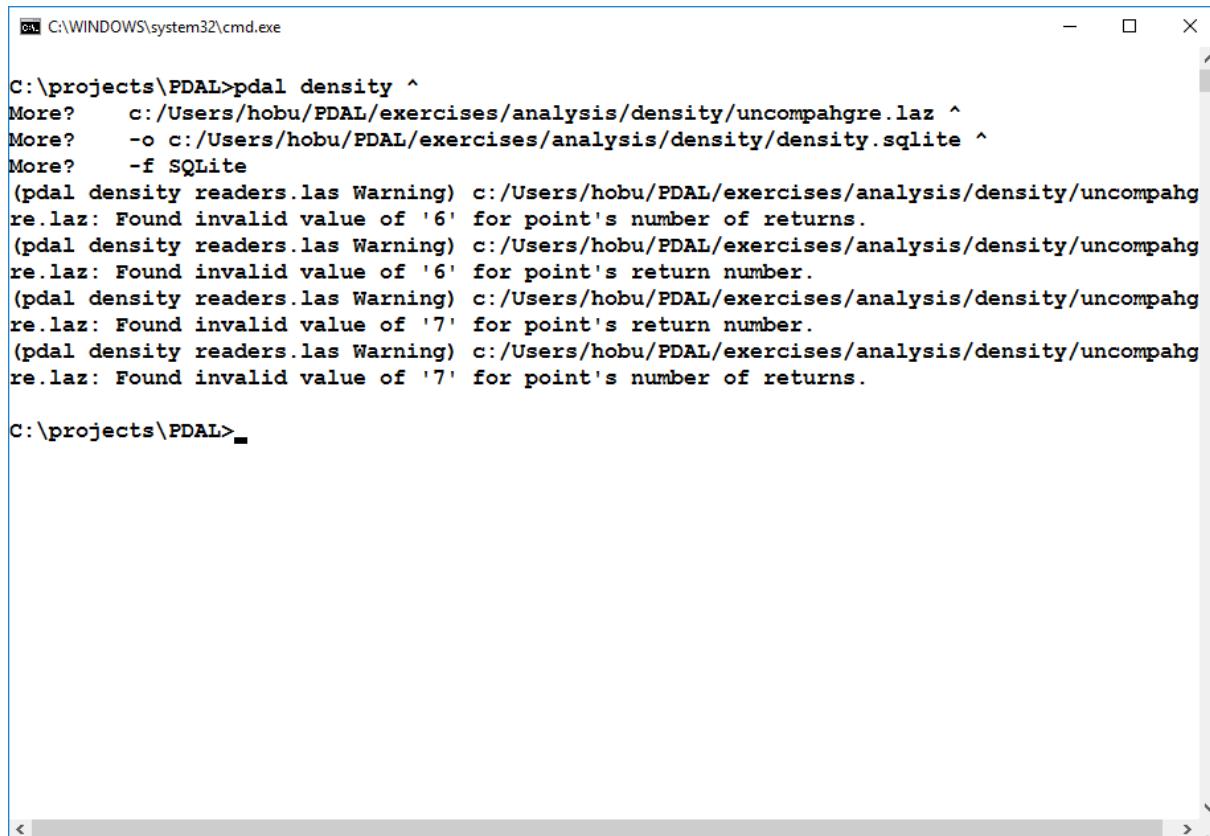
Exercise

PDAL provides an `application` (page 25) to compute a vector field of hexagons computed with `filters.hexbin` (page 134). 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 *OSGeo4W Shell*:

```
1 pdal density ^
2   c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz ^
3   -o c:/Users/hobu/PDAL/exercises/analysis/density/density.sqlite ^
4   -f SQLite
```



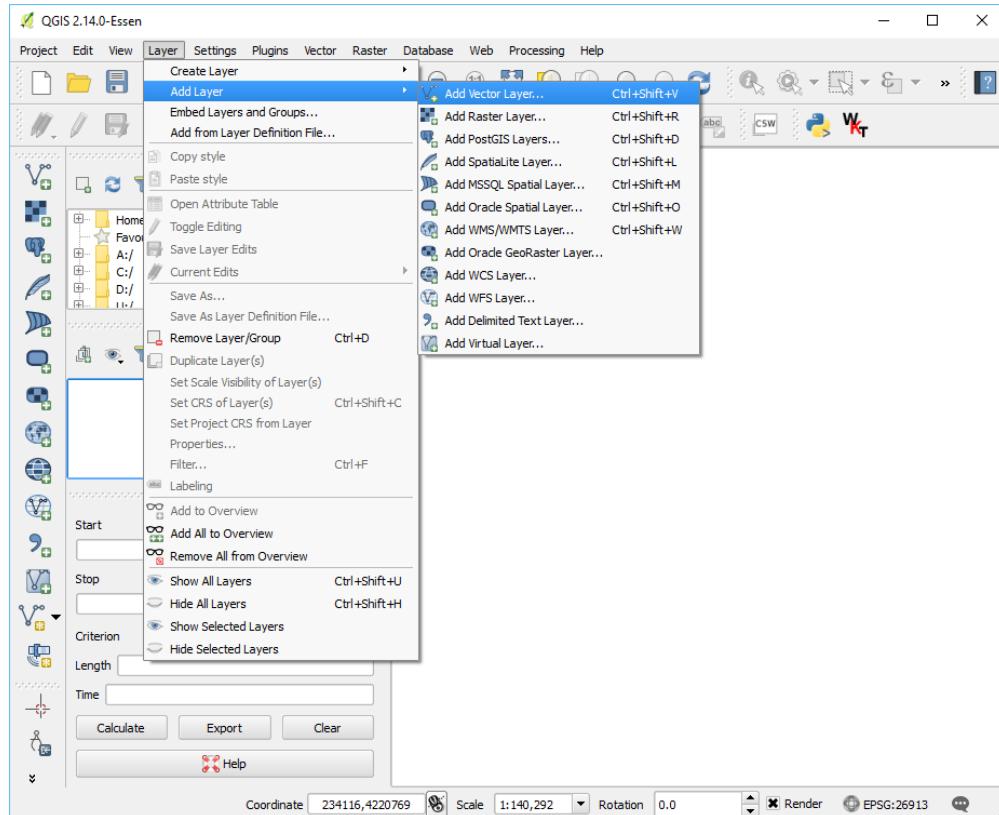
```
C:\projects\PDAL>pdal density ^
More?   c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz ^
More?   -o c:/Users/hobu/PDAL/exercises/analysis/density/density.sqlite ^
More?   -f SQLite
(pdal density readers.las Warning) c:/Users/hobu/PDAL/exercises/analysis/density/uncompahg
re.laz: Found invalid value of '6' for point's number of returns.
(pdal density readers.las Warning) c:/Users/hobu/PDAL/exercises/analysis/density/uncompahg
re.laz: Found invalid value of '6' for point's return number.
(pdal density readers.las Warning) c:/Users/hobu/PDAL/exercises/analysis/density/uncompahg
re.laz: Found invalid value of '7' for point's return number.
(pdal density readers.las Warning) c:/Users/hobu/PDAL/exercises/analysis/density/uncompahg
re.laz: Found invalid value of '7' for point's number of returns.

C:\projects\PDAL>_
```

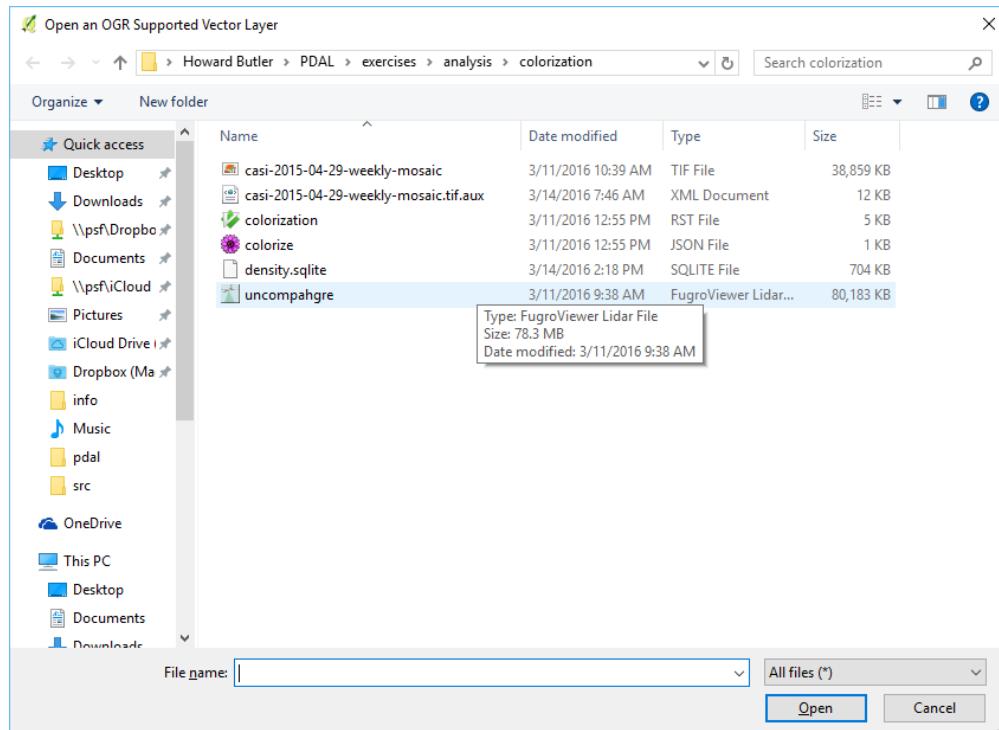
Visualization

The command uses GDAL to output a [SQLite](http://sqlite.org) (<http://sqlite.org>) file containing hexagon polygons. We will now use [QGIS](http://qgis.org) (<http://qgis.org>) to visualize them.

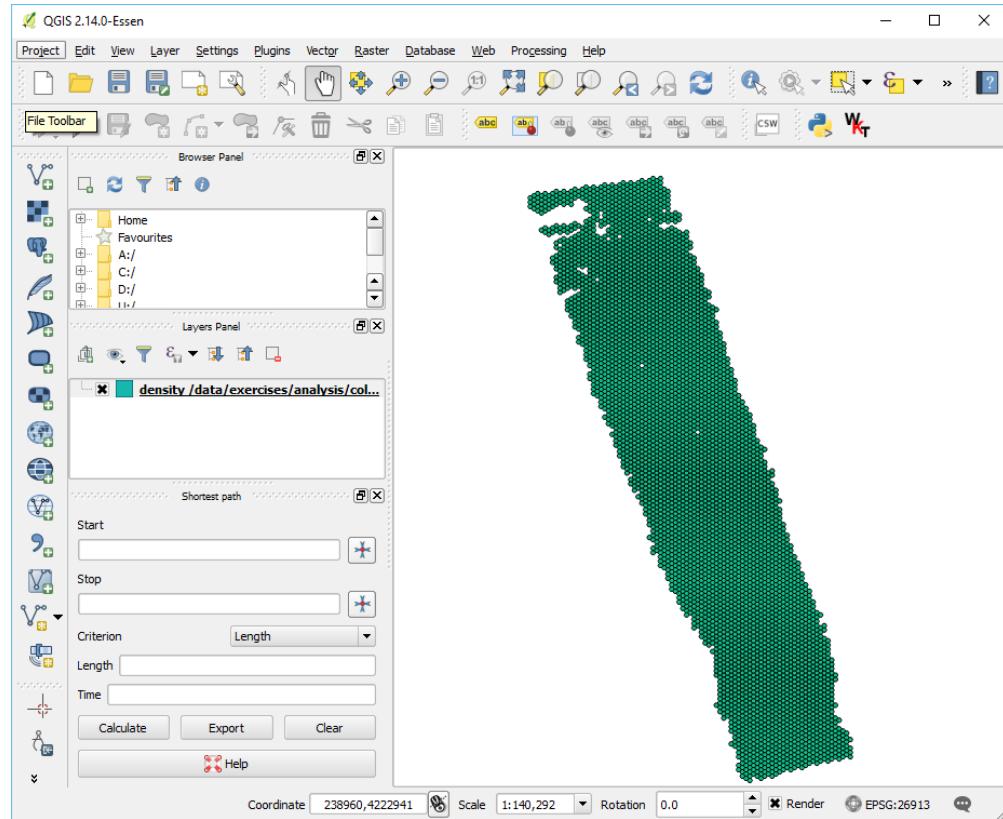
1. Add a vector layer



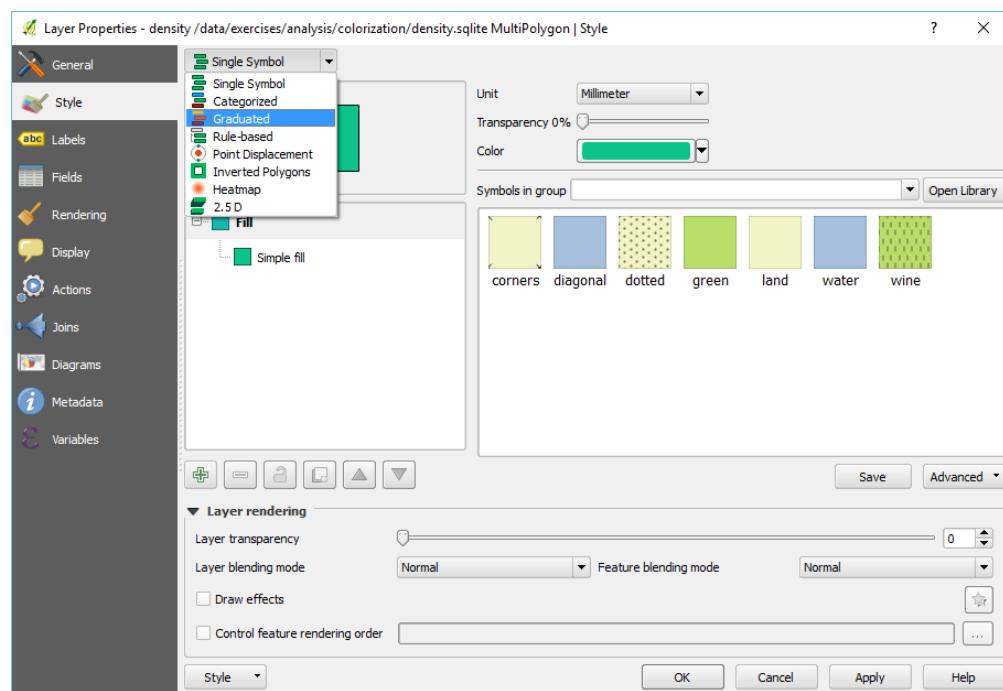
2. Navigate to the output directory



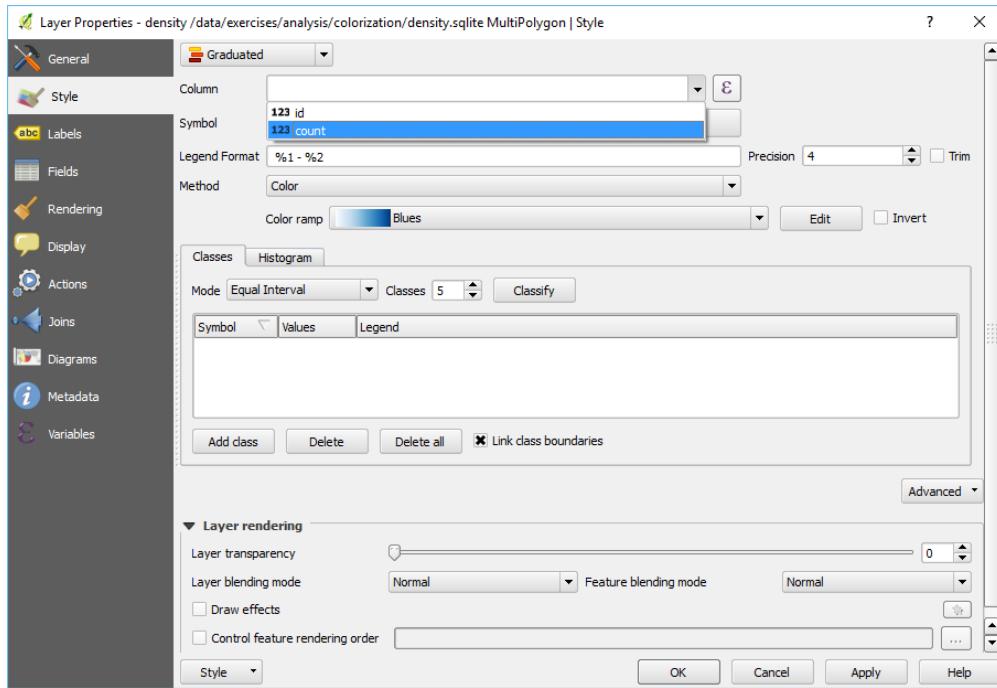
3. Add the density.sqlite file to the view



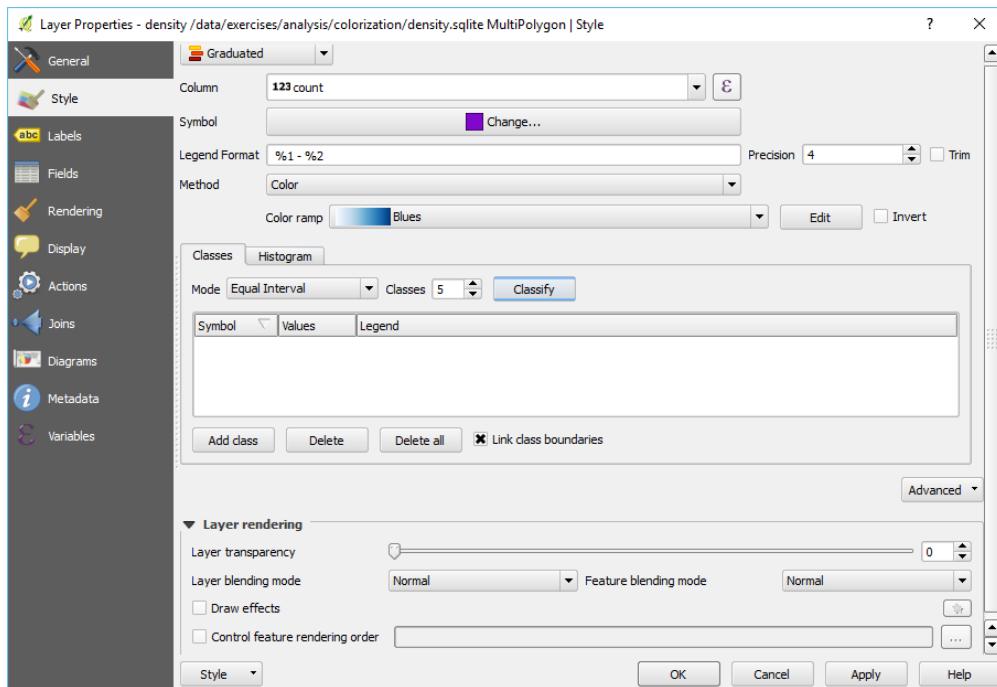
4. Right click on the density.sqlite layer in the *Layers* panel and then choose *Properties*.
5. Pick the *Graduated* drop down



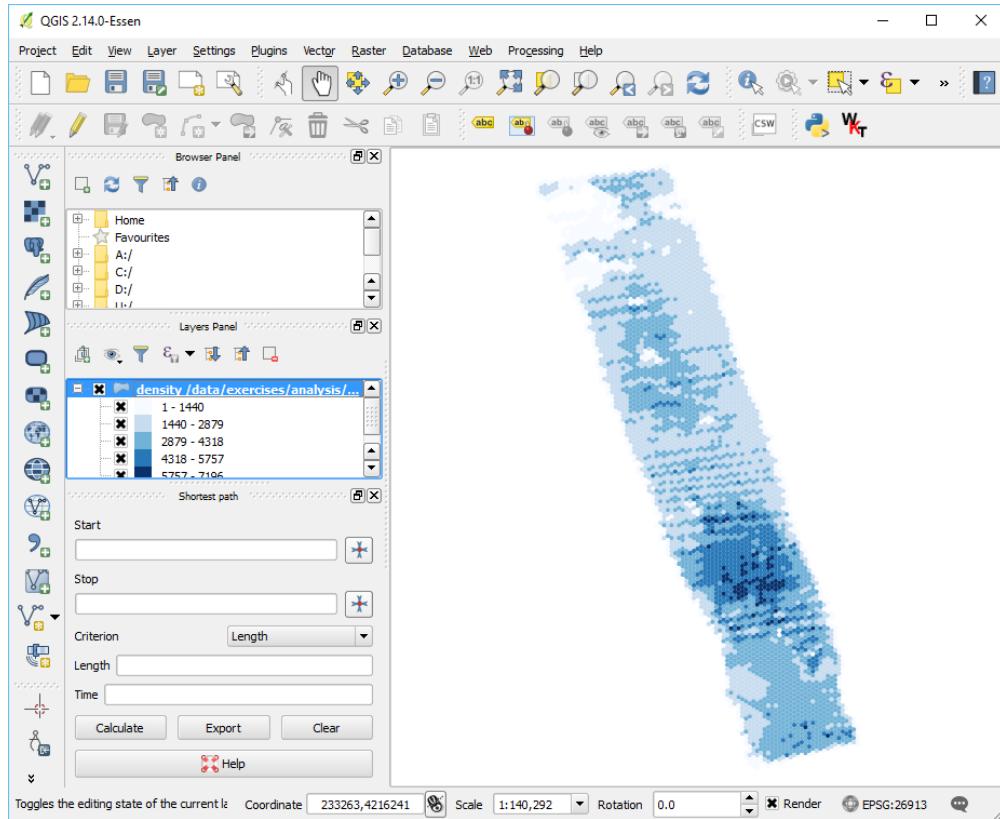
6. Choose the Count column to visualize



7. Choose the Classify button to add intervals



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 134).

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 34).

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.

Note: This exercise is an adaptation of the [Performing Poisson Sampling of Point Clouds Using Dart Throwing](#) (page 227) tutorial on the PDAL website by Brad Chambers. It includes

some images from that tutorial for illustration. You can find more detail and example invocations there.

Exercise

As we showed in the [Visualizing acquisition density](#) (page 282) 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 175)
- Random using a combination of [*filters.decimation*](#) (page 119) and [*filters.randomize*](#) (page 170)
- Voxel using [*filters.voxelgrid*](#) (page 184)

In this exercise, we are going to thin with the Poisson method, but the concept should operate similarly for the [*filters.voxelgrid*](#) (page 184) approach too. See [Performing Poisson Sampling of Point Clouds Using Dart Throwing](#) (page 227) for description of how to randomly filter.

Command

Invoke the following command, substituting accordingly, in your *Docker Quickstart Terminal*:

```
1 pdal translate ^
2   c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz ^
3   c:/Users/hobu/PDAL/exercises/analysis/thinning/uncompahgre-thin.
4   ↵laz ^
5     sample ^
6       --filters.sample.radius=20
```

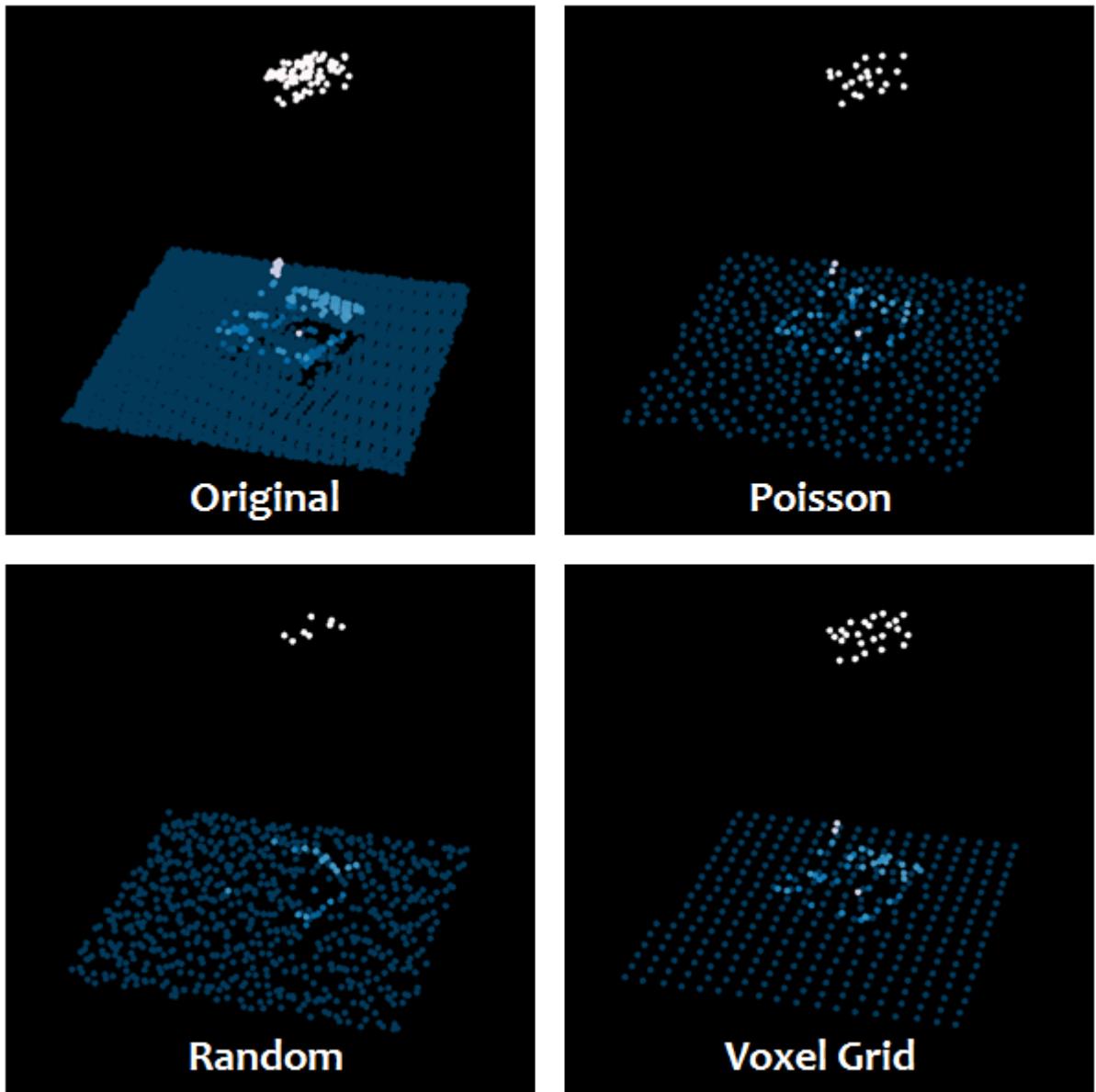
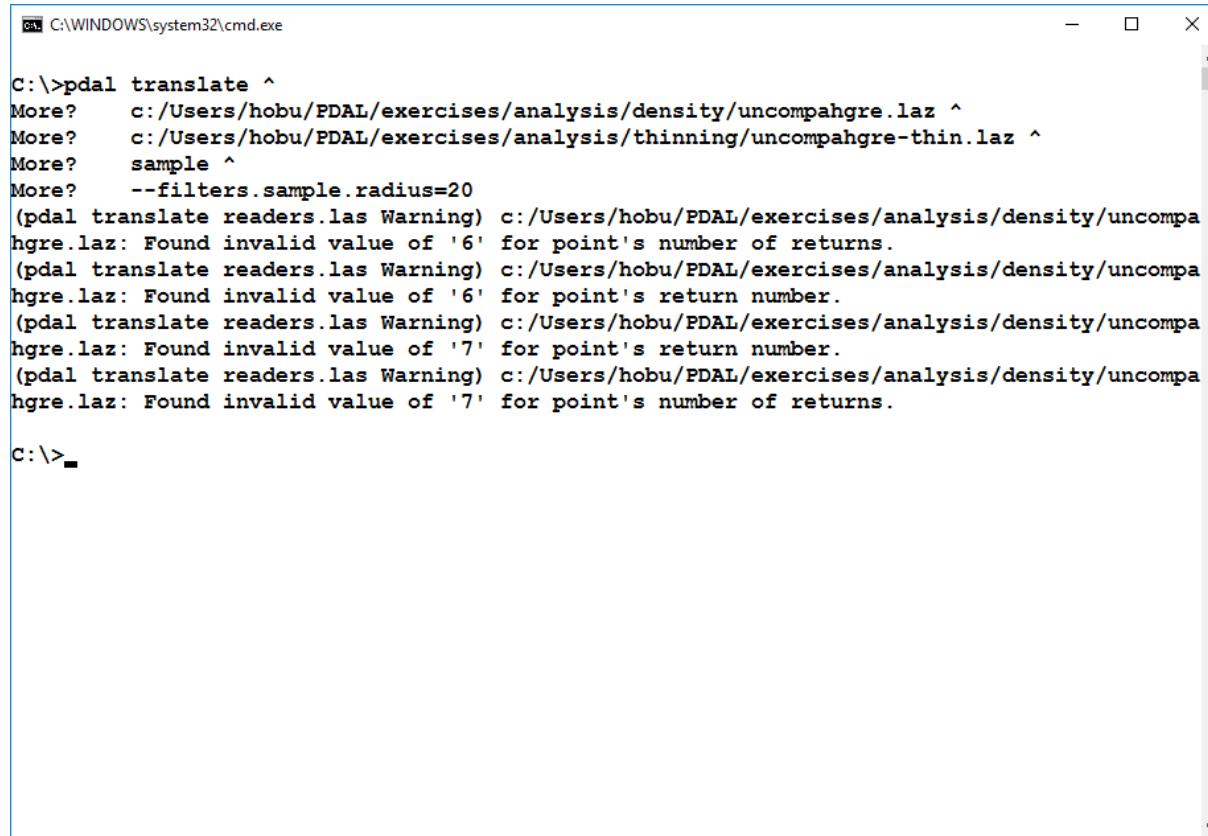


Fig. 11.3: Thinning strategies available in PDAL



The screenshot shows a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The command entered is 'pdal translate ^ More? c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz ^ More? c:/Users/hobu/PDAL/exercises/analysis/thinning/uncompahgre-thin.laz ^ More? sample ^ More? --filters.sample.radius=20 (pdal translate readers.las Warning) c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz: Found invalid value of '6' for point's number of returns. (pdal translate readers.las Warning) c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz: Found invalid value of '6' for point's return number. (pdal translate readers.las Warning) c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz: Found invalid value of '7' for point's return number. (pdal translate readers.las Warning) c:/Users/hobu/PDAL/exercises/analysis/density/uncompahgre.laz: Found invalid value of '7' for point's number of returns.' The command ends with 'C:\>-'.

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.

Notes

1. Poisson sampling is non-destructive. Points that are filtered with *filters.sample* (page 175) 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 *Identifying ground returns using ProgressiveMorphologicalFilter segmentation* (page 214) tutorial on the PDAL website by Brad Chambers. You can find more detail and example invocations there.

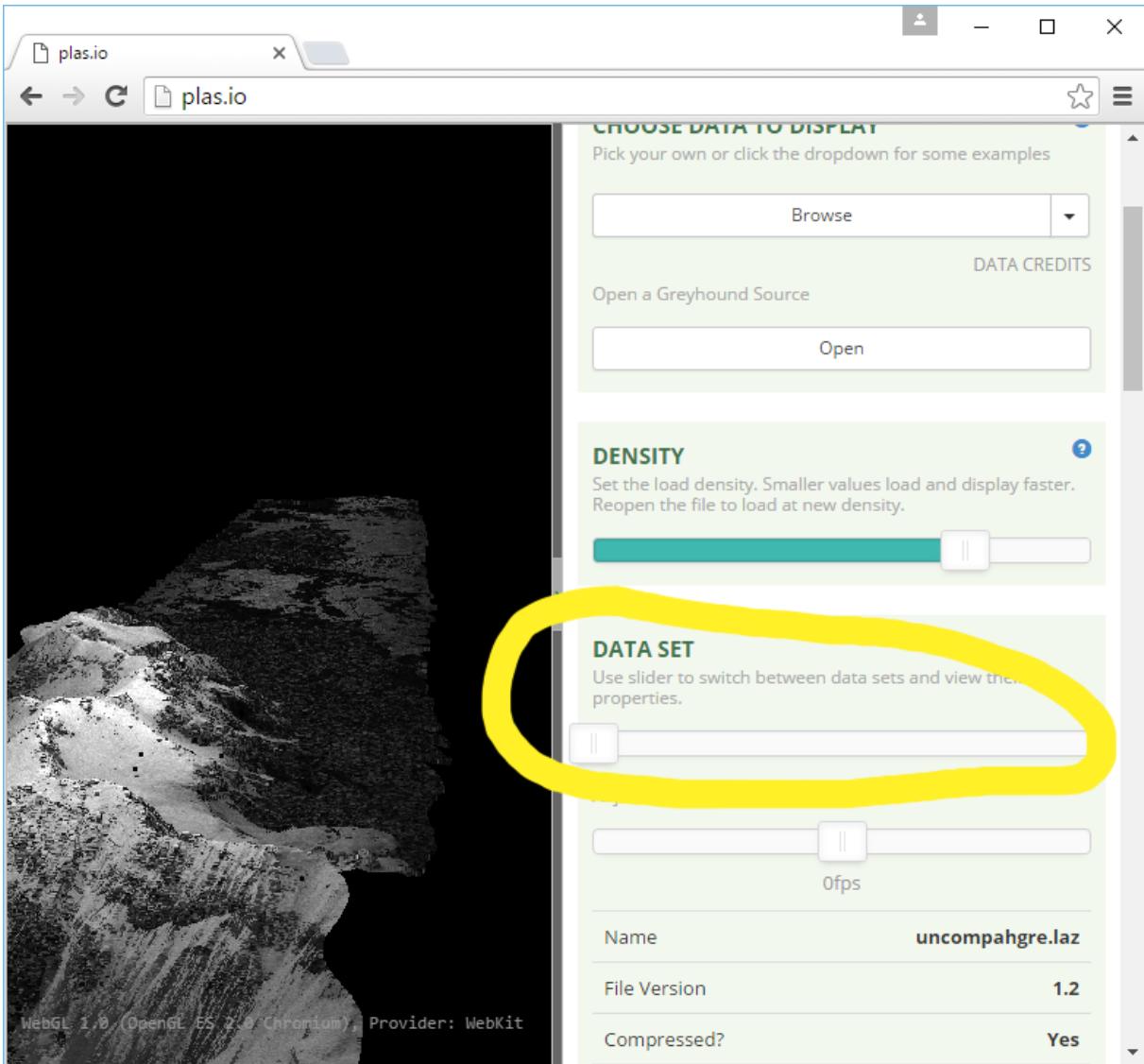


Fig. 11.4: Selecting multiple data sets in <http://plas.io>

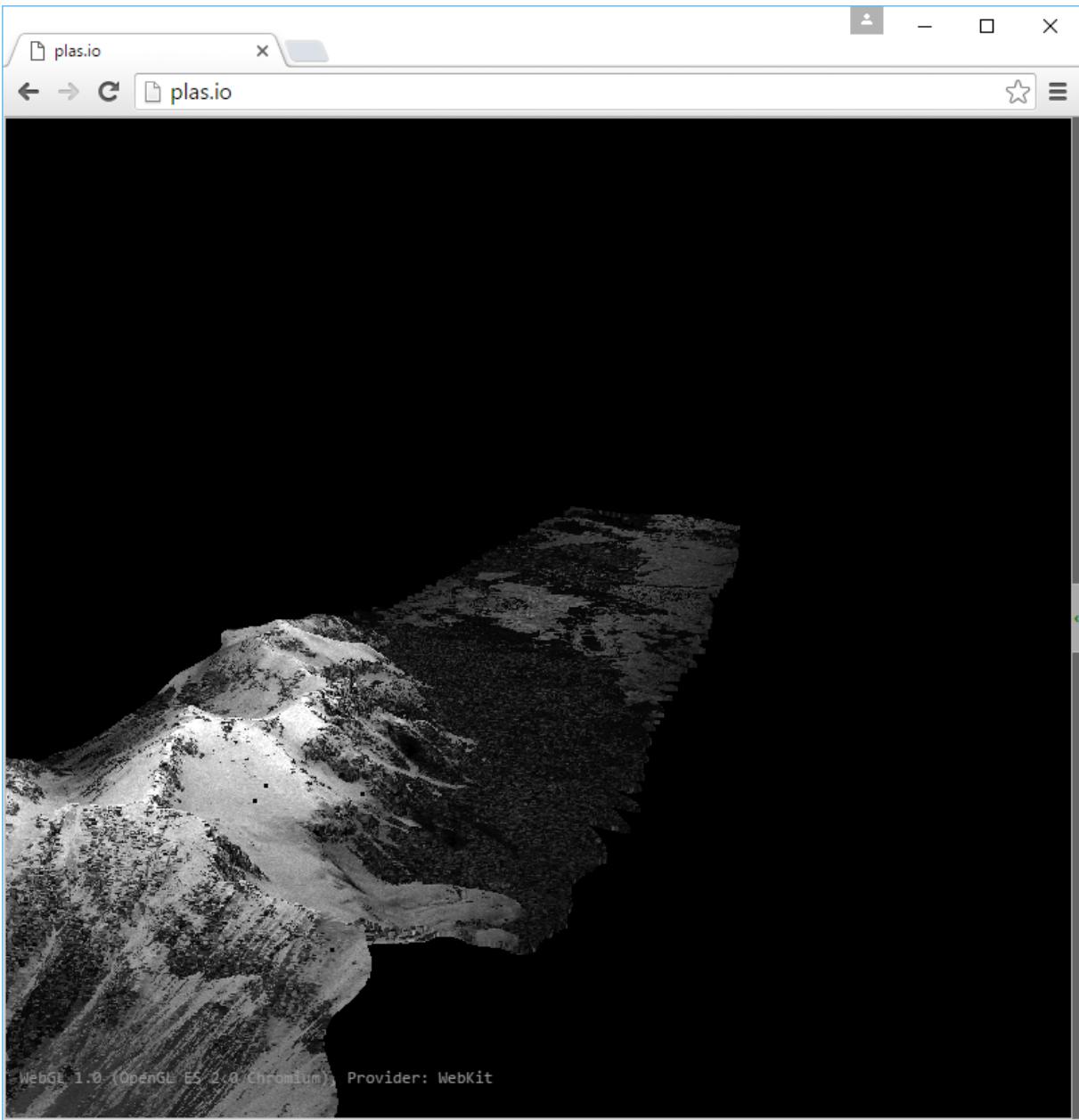


Fig. 11.5: Full resolution Uncompahgre data set

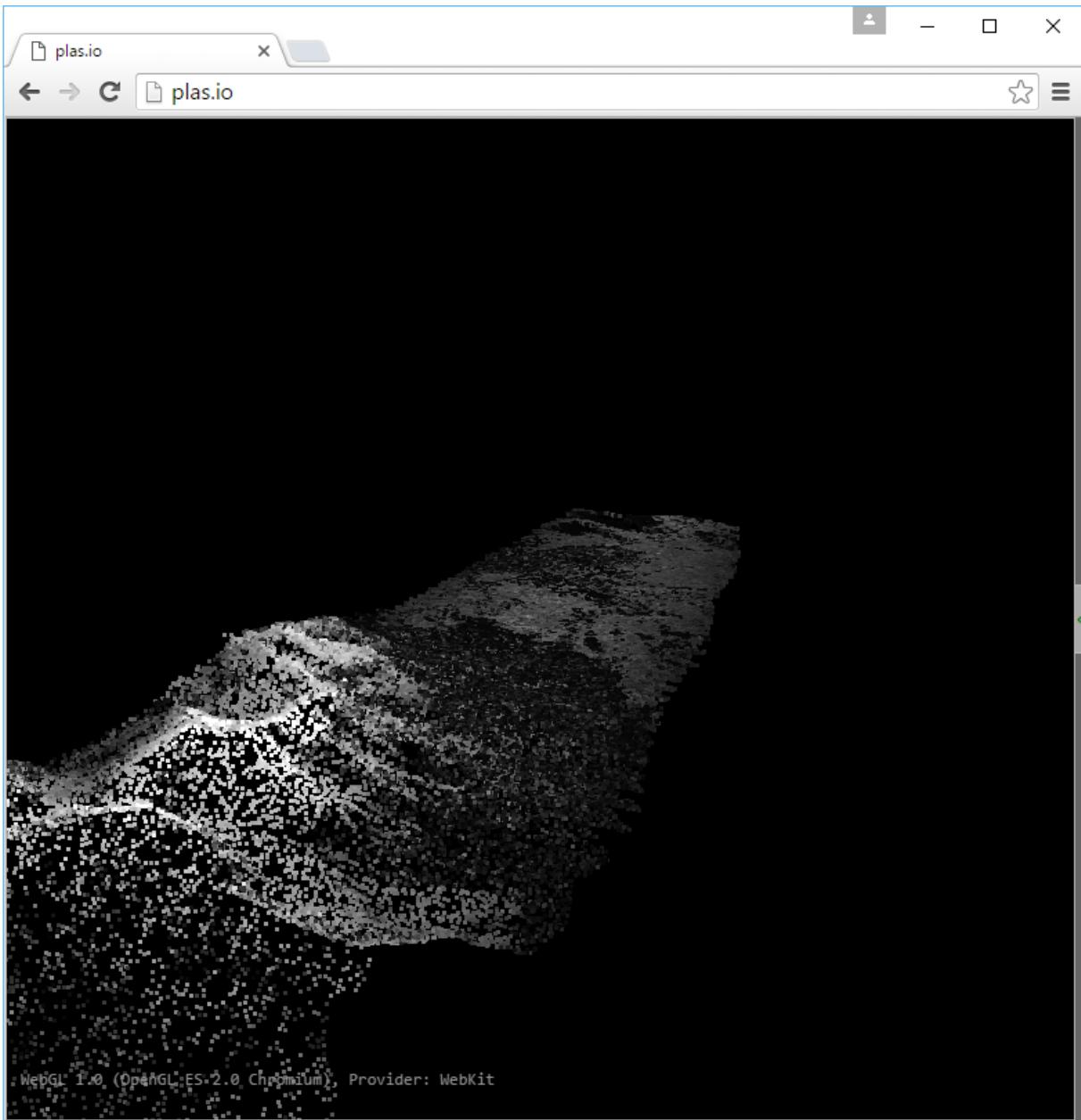


Fig. 11.6: Uncompahgre thinned at a radius of 20m

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].⁴

Command

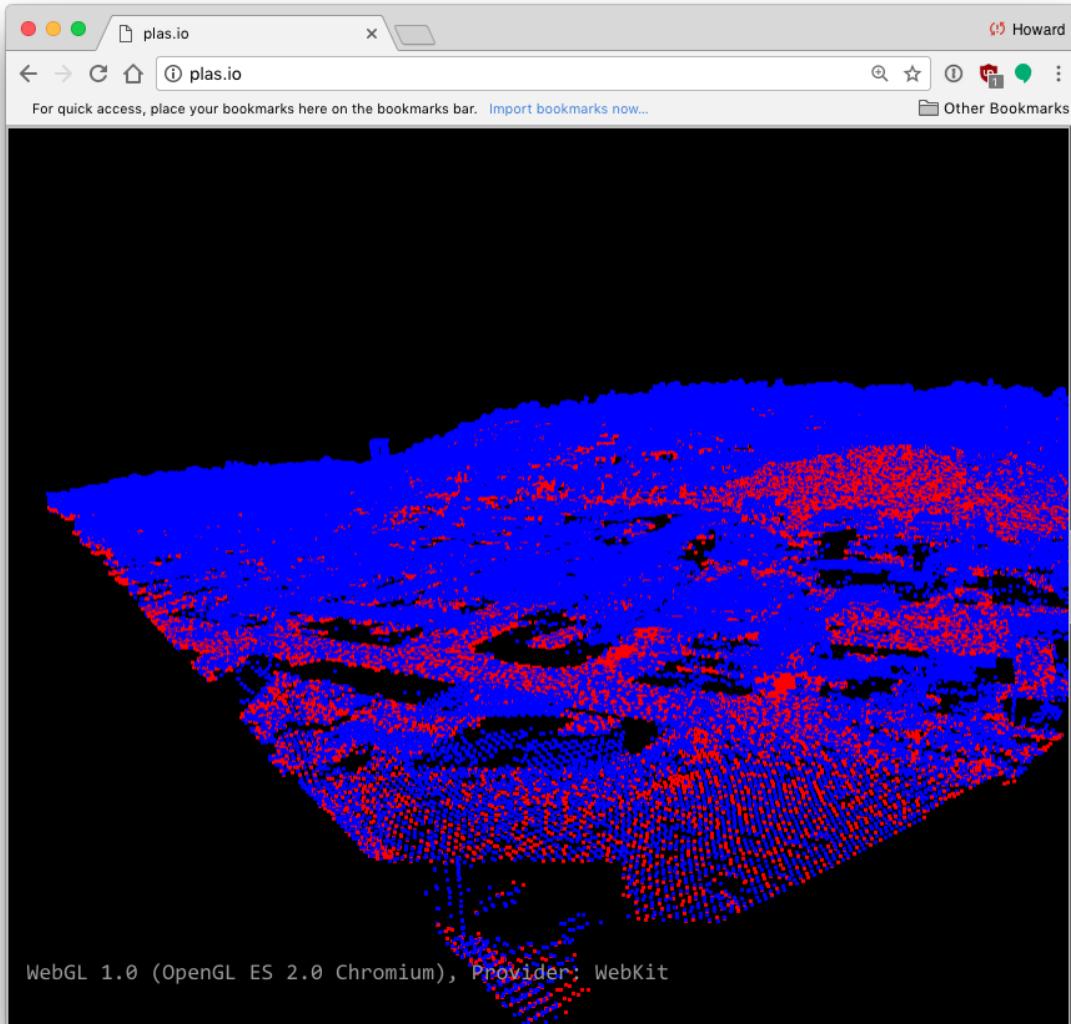
Invoke the following command, substituting accordingly, in your *OSGeo4W Shell*:

```
1 pdal translate ^
2   c:/Users/hobu/PDAL/exercises/analysis/ground/CSite1_orig-utm.laz ^
3   -o c:/Users/hobu/PDAL/exercises/analysis/ground/ground.laz ^
4   smrf ^
5   -v 4
6
```

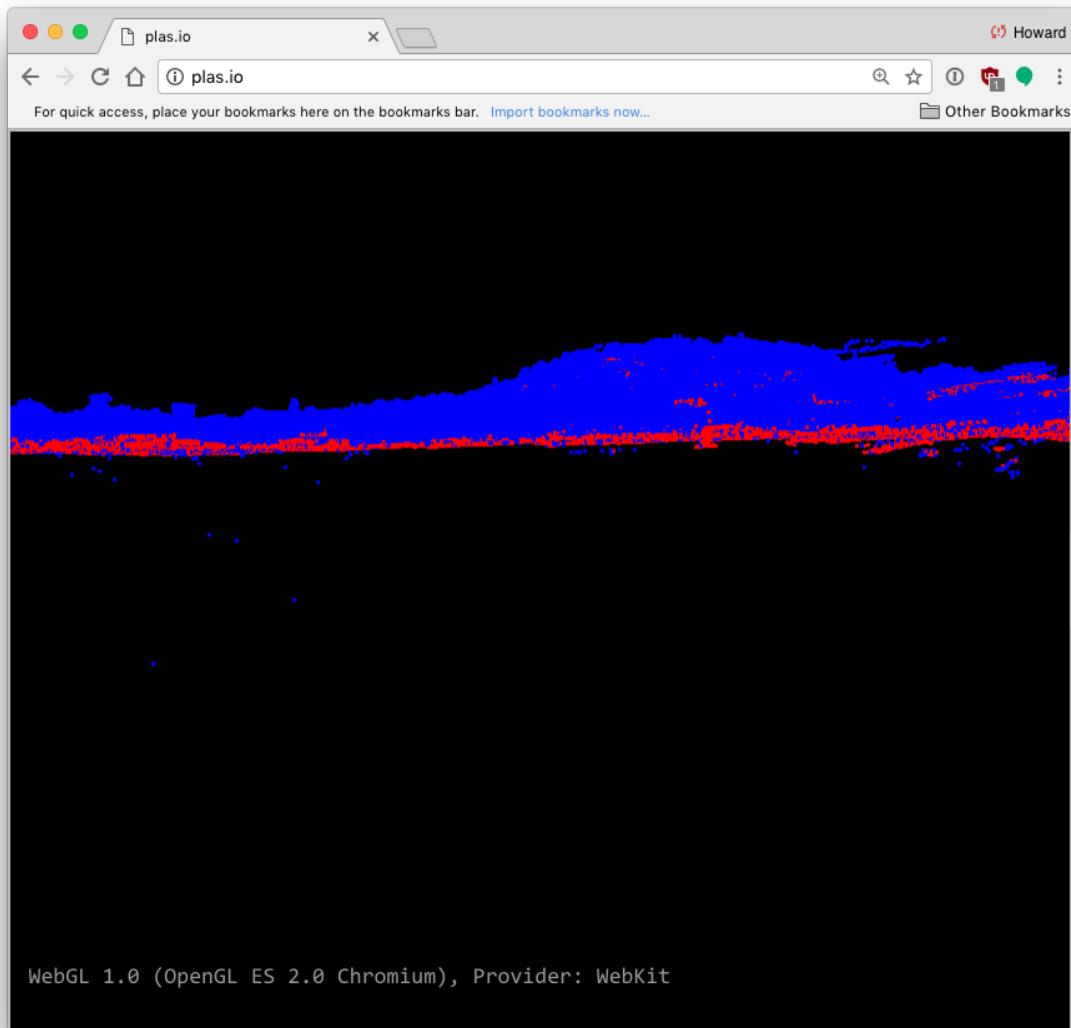
```
C:\>pdal translate ^
More?   c:/Users/hobu/PDAL/exercises/analysis/ground/CSite1_orig-utm.laz ^
More?   -o c:/Users/hobu/PDAL/exercises/analysis/ground/ground.laz ^
More?   smrf ^
More?   -v 4
(pdal translate filters.smrf Debug) progressiveFilter: radius = 1      767105 ground  46
96 non-ground (0.61%)
(pdal translate filters.smrf Debug) progressiveFilter: radius = 1      576486 ground  19
5315 non-ground (25.31%)
(pdal translate filters.smrf Debug) progressiveFilter: radius = 2      518852 ground  25
2949 non-ground (32.77%)
(pdal translate filters.smrf Debug) progressiveFilter: radius = 3      491808 ground  27
9993 non-ground (36.28%)
(pdal translate filters.smrf Debug) progressiveFilter: radius = 4      473005 ground  29
8796 non-ground (38.71%)
(pdal translate filters.smrf Debug) progressiveFilter: radius = 5      453949 ground  31
7852 non-ground (41.18%)
(pdal translate filters.smrf Debug) progressiveFilter: radius = 6      433190 ground  33
8611 non-ground (43.87%)
(pdal translate filters.smrf Debug) progressiveFilter: radius = 7      414495 ground  35
7306 non-ground (46.30%)
(pdal translate filters.smrf Debug) progressiveFilter: radius = 8      399673 ground  37
2128 non-ground (48.22%)
(pdal translate filters.smrf Debug) progressiveFilter: radius = 9      387931 ground  38
3870 non-ground (49.74%)
(pdal translate filters.smrf Debug) progressiveFilter: radius = 10     382948 ground  38
8853 non-ground (50.38%)
(pdal translate filters.smrf Debug) progressiveFilter: radius = 11     379436 ground  39
```

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 PMF with a call to a the *filters.outlier* technique we learned about in denoising.

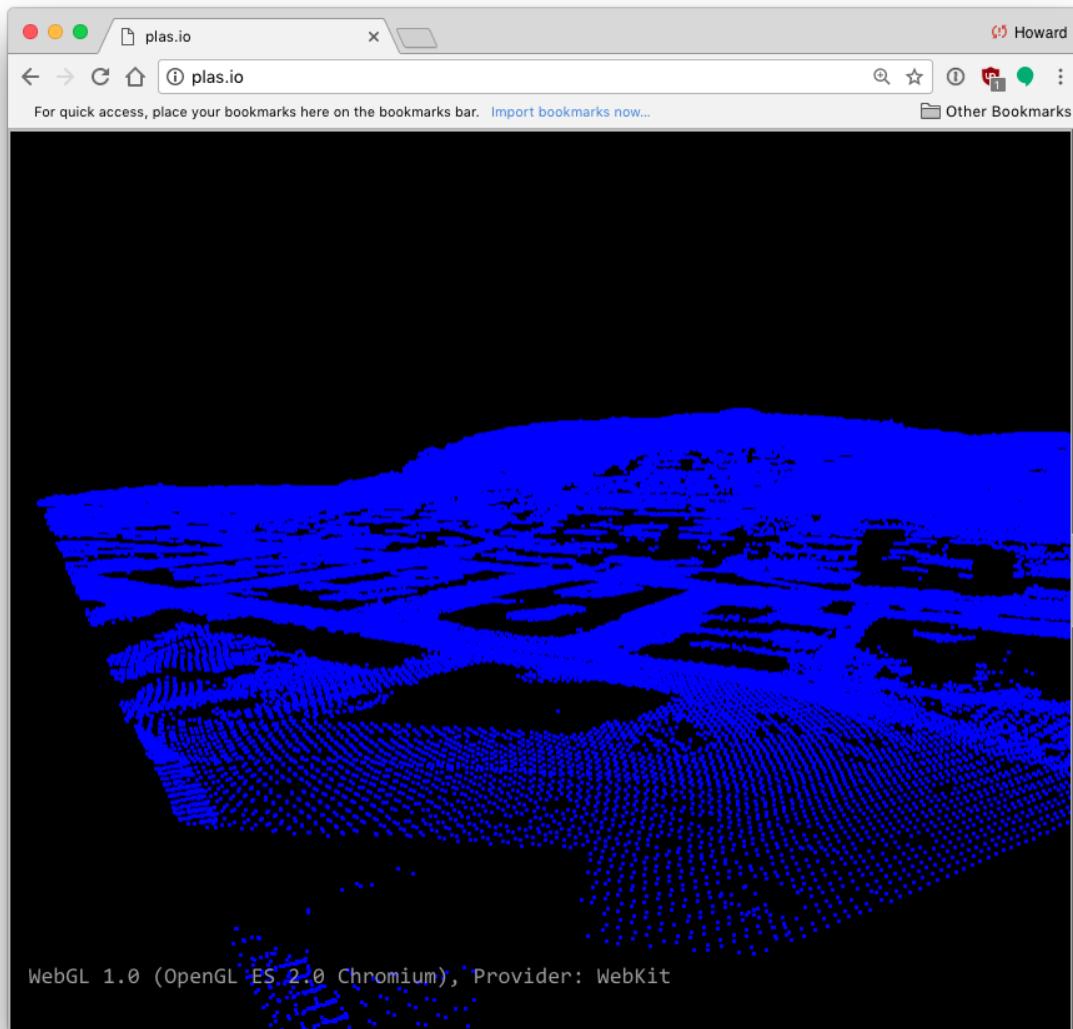
1. Let us start by removing the non-ground data:

```
1 pdal translate ^
  c:/Users/hobu/PDAL/exercises/analysis/ground/CSitel_orig-utm.laz ^
  -o c:/Users/hobu/PDAL/exercises/analysis/ground/ground.laz ^
  smrf ^
  range ^
```

```

6   --filters.range.limits="Classification[2:2]" ^
7   -v 4
8

```



2. Now we will instead use the *translate* (page 36) command to stack the *filters.outlier* (page 151) and *filters.smrf* (page 176) stages:

```

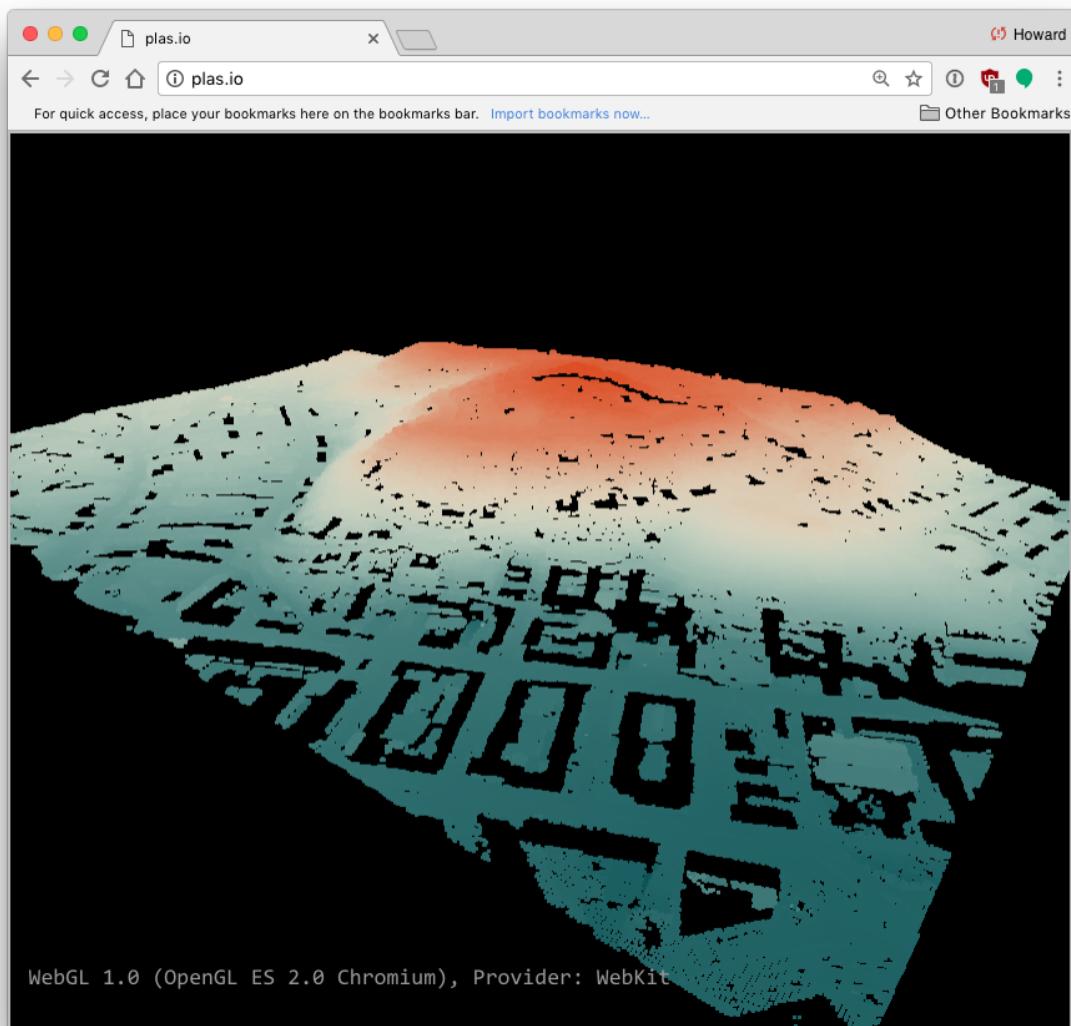
1 pdal translate ^
2   c:/Users/hobu/PDAL/exercises/analysis/ground/CSitel_orig-utm.laz ^
3   -o c:/Users/hobu/PDAL/exercises/analysis/ground/denoised-ground-
4   ↵only.laz ^
5     outlier smrf range ^
6       --filters.outlier.method="statistical" ^
7       --filters.outlier.mean_k=8 ^
8

```

```
7 --filters.outlier.multiplier=3.0 ^
8 --filters.smrf.ignore="Classification[7:7]" ^
9 --filters.range.limits="Classification[2:2]" ^
10 --writers.las.compression=true --verbose 4
```

In this invocation, we have more control over the process. First the outlier filter merely classifies outliers with a Classification value of 7. These outliers are then ignored during PMF processing with the ignore option. Finally, we add a range filter to extract only the ground returns (i.e., 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 ground exercise, PDAL's [writers.gdal](#) (page 81) 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 ground exercise. Please produce that file by following that exercise before starting this one.

Command

Invoke the following command, substituting accordingly, in your *OSGeo4W Shell*:

PDAL capability to generate rasterized output is provided by the [writers.gdal](#) (page 81) stage. There is no [application](#) (page 23) to drive this stage, and we must use a pipeline.

Pipeline breakdown

```
{  
    "pipeline": [  
        "c:/Users/hobu/PDAL/exercises/analysis/ground/denoised-  
        ↪ground-only.laz",  
        {  
            "filename": "c:/Users/hobu/PDAL/exercises/analysis/dtm/  
            ↪dtm.tif",  
            "output_format": "tif",  
            "output_type": "all",  
            "grid_dist_x": "2.0",  
            "grid_dist_y": "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

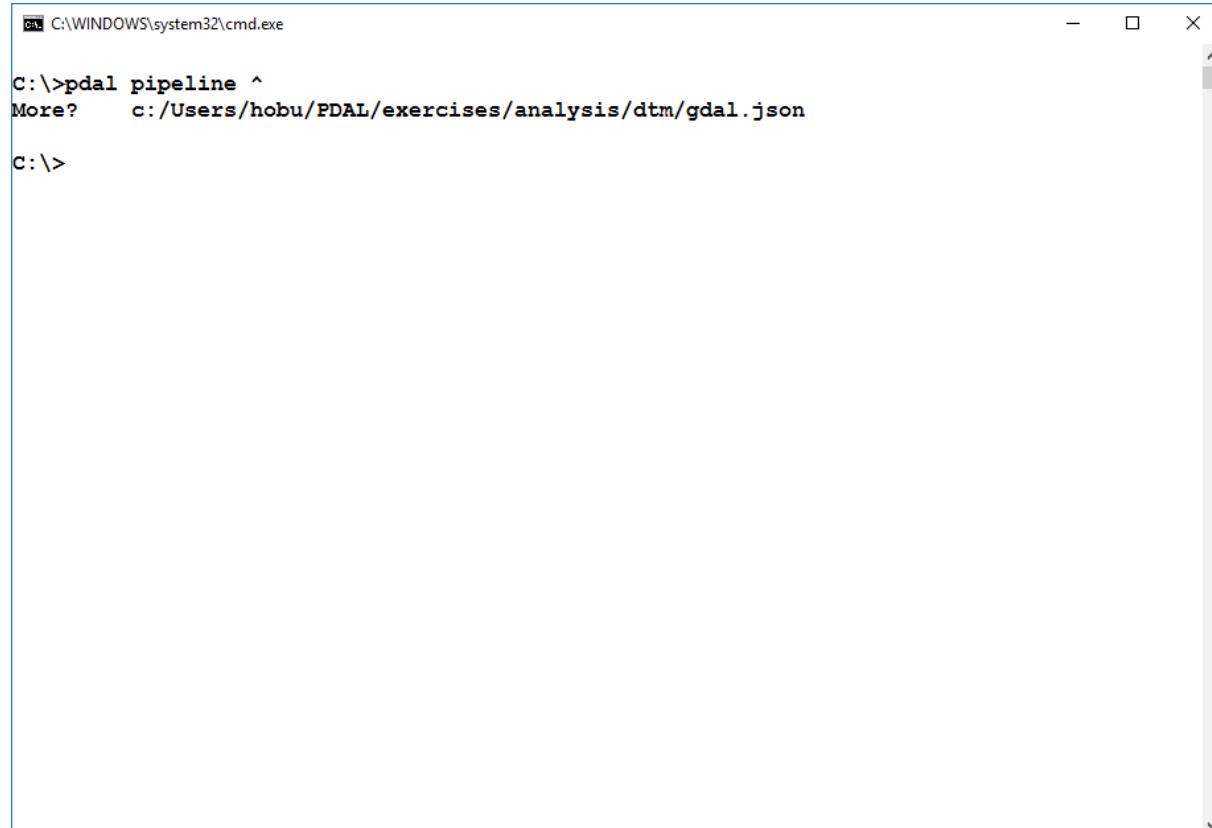
denoised-ground-only is the [LASzip](http://laszip.org) (<http://laszip.org>) file we will clip. You should have created this output as part of the ground exercise.

2. writers.gdal

The [*writers.gdal*](#) (page 81) writer that bins the point cloud data into an elevation surface.

Execution

```
1 pdal pipeline ^
2   c:/Users/hobu/PDAL/exercises/analysis/dtm/gdal.json
```

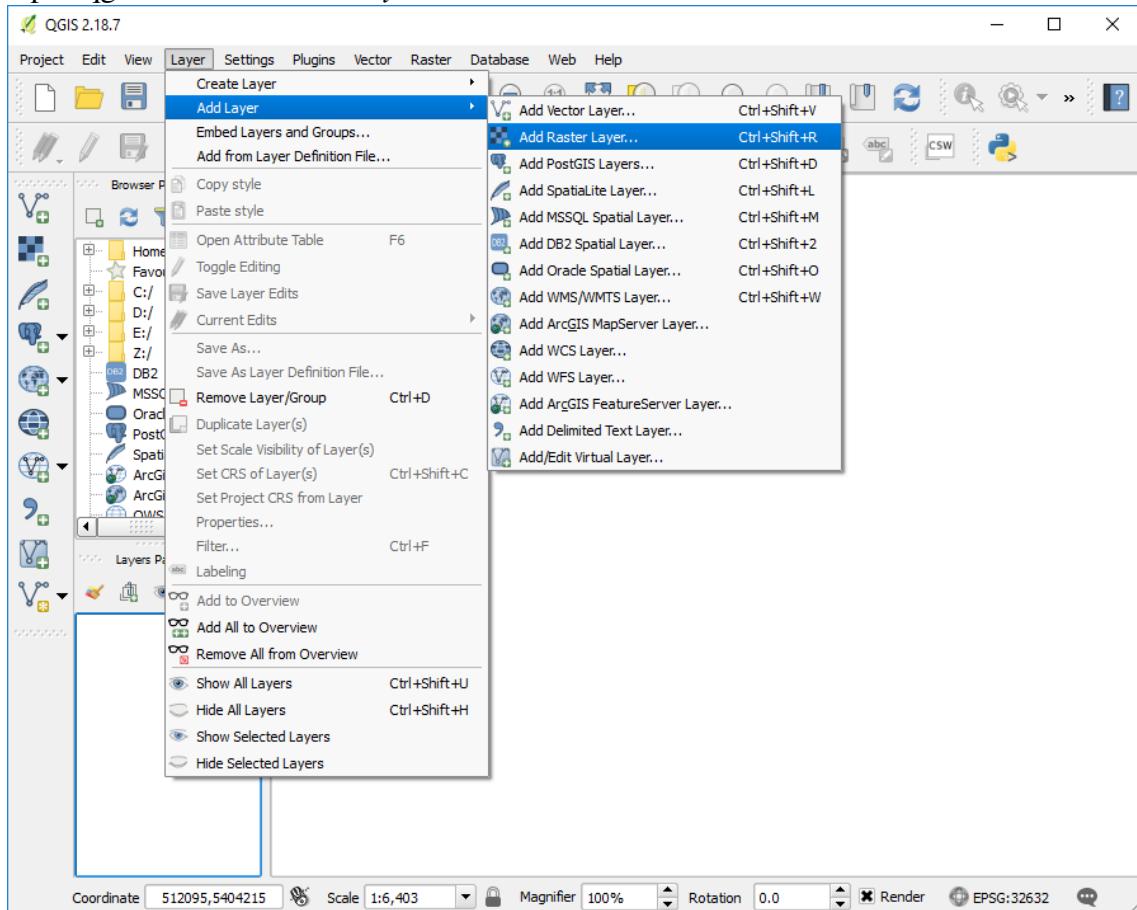


A screenshot of a Windows command prompt window titled "cmd C:\WINDOWS\system32\cmd.exe". The window shows the command "pdal pipeline ^" followed by a file path "c:/Users/hobu/PDAL/exercises/analysis/dtm/gdal.json". The command is partially typed, with the cursor at the end of "gdal.json". The window has standard minimize, maximize, and close buttons in the top right corner.

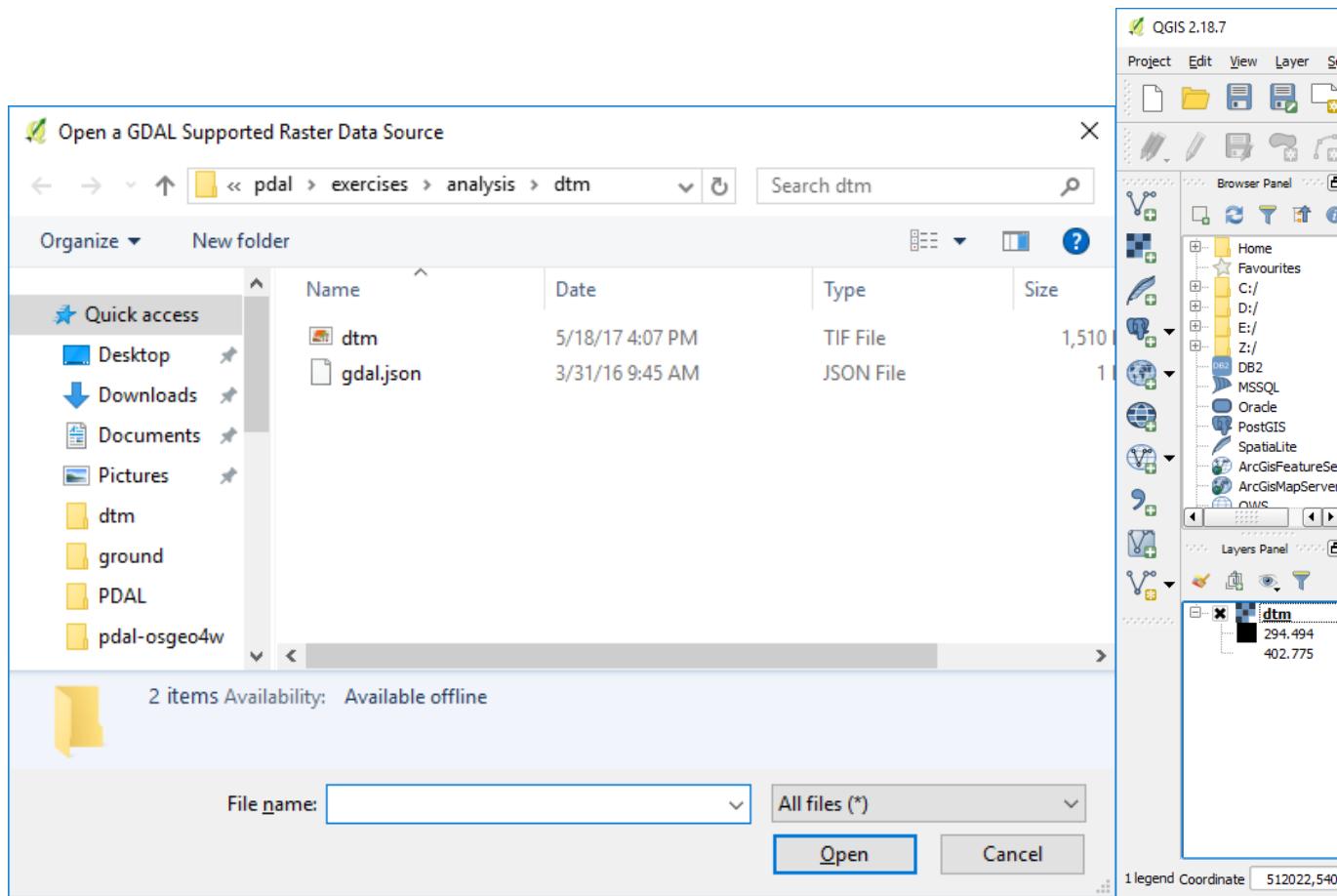
Visualization

Something happened, and some files were written, but we cannot really see what was produced. Let us use qgis to visualize the output.

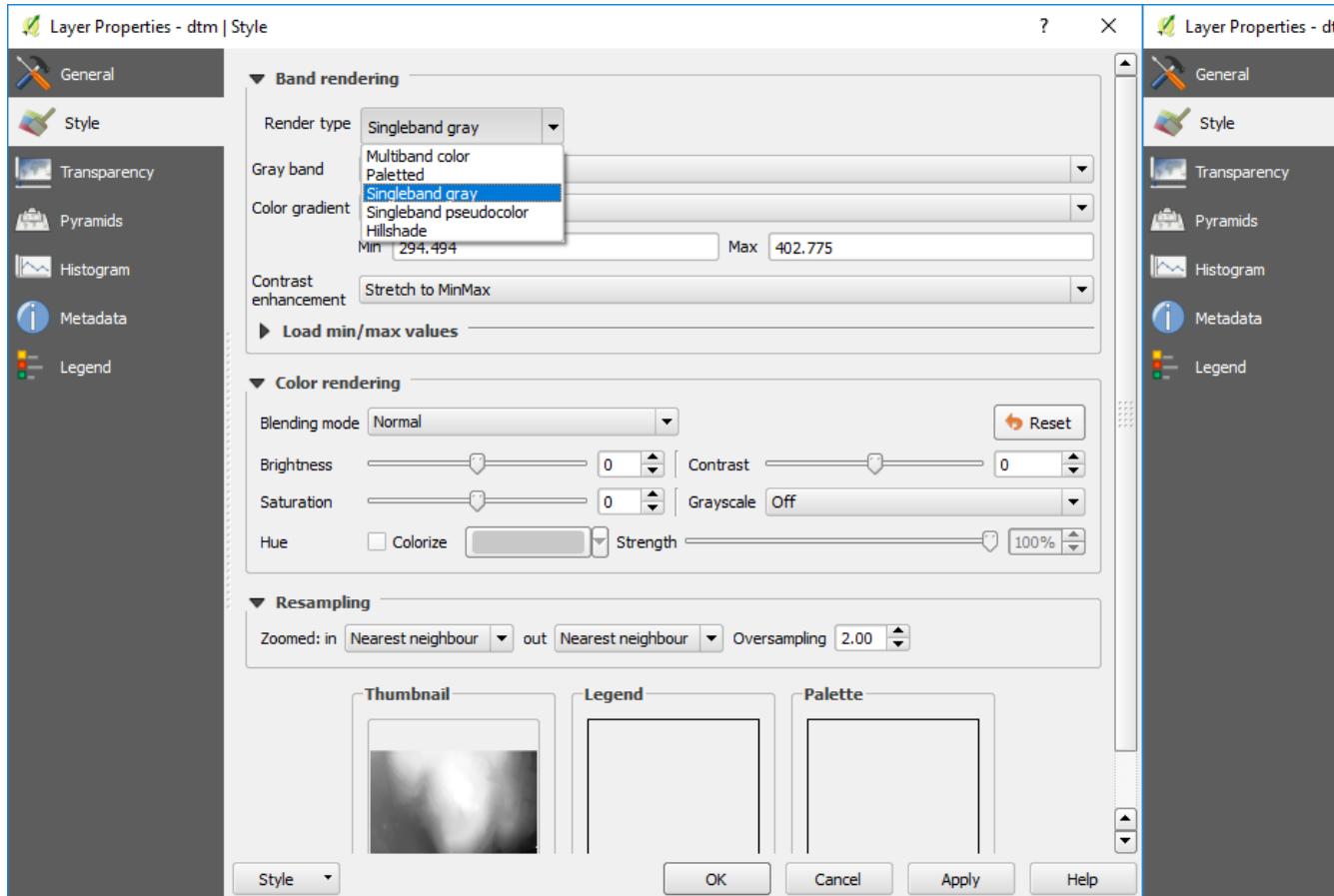
1. Open qgis and *Add Raster Layer*:



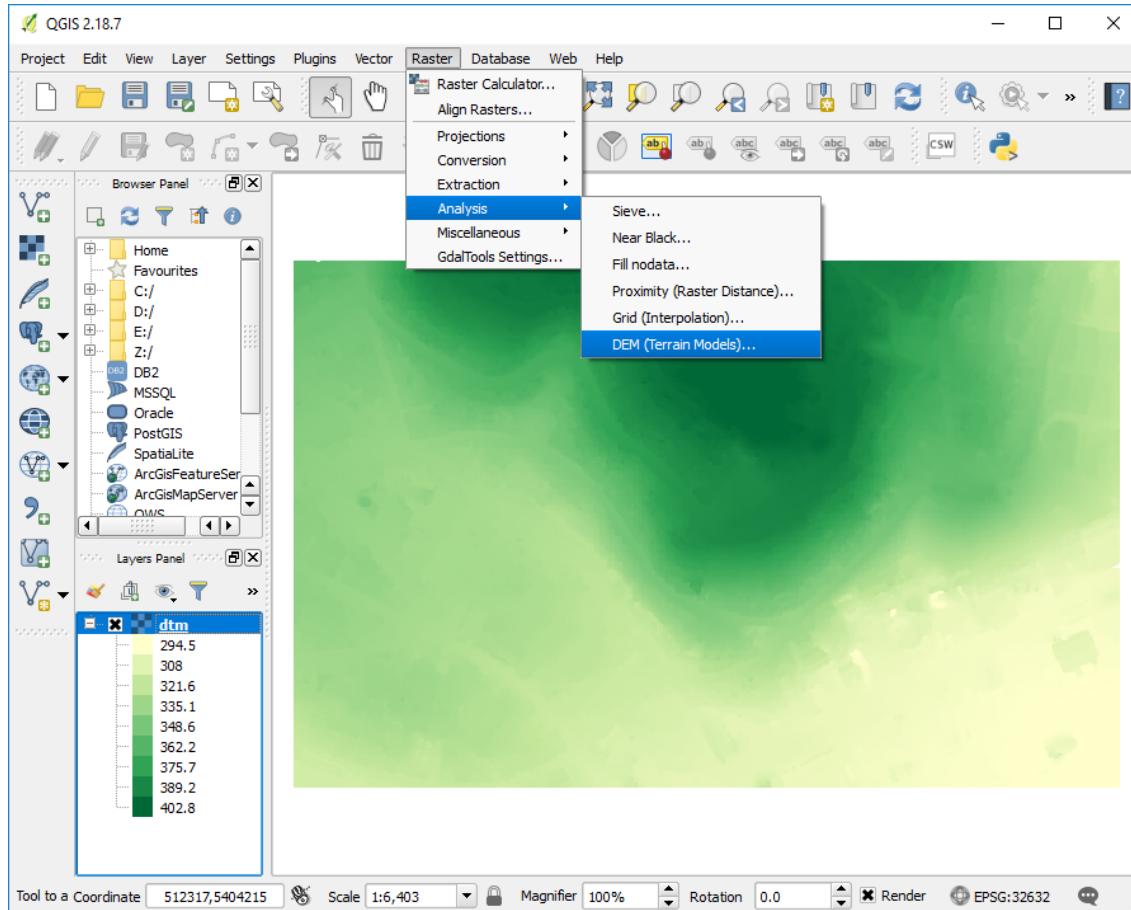
2. Add the *dtm.tif* file from your `./PDAL/exercises/analysis/dtm` directory.



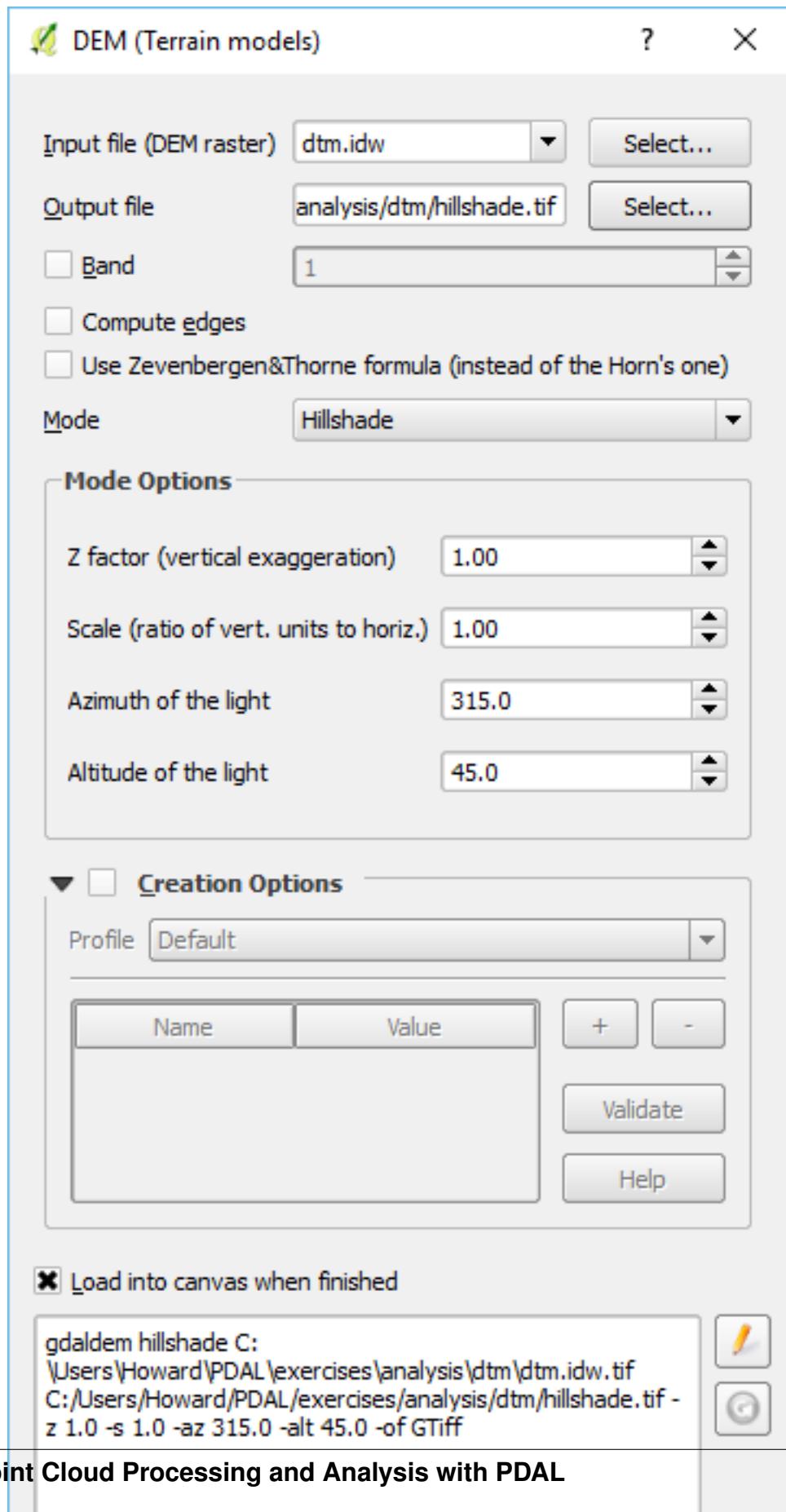
3. Classify the DTM by right-clicking on the *dtm.tif* and choosing *Properties*. Pick the pseudocolor rendering type, and then choose a color ramp and click *Classify*.



4. qgis provides access to [GDAL](http://gdal.org/) (<http://gdal.org/>) processing tools, and we are going to use that to create a hillshade of our surface. Choose *Raster->Analysis->Dem*:

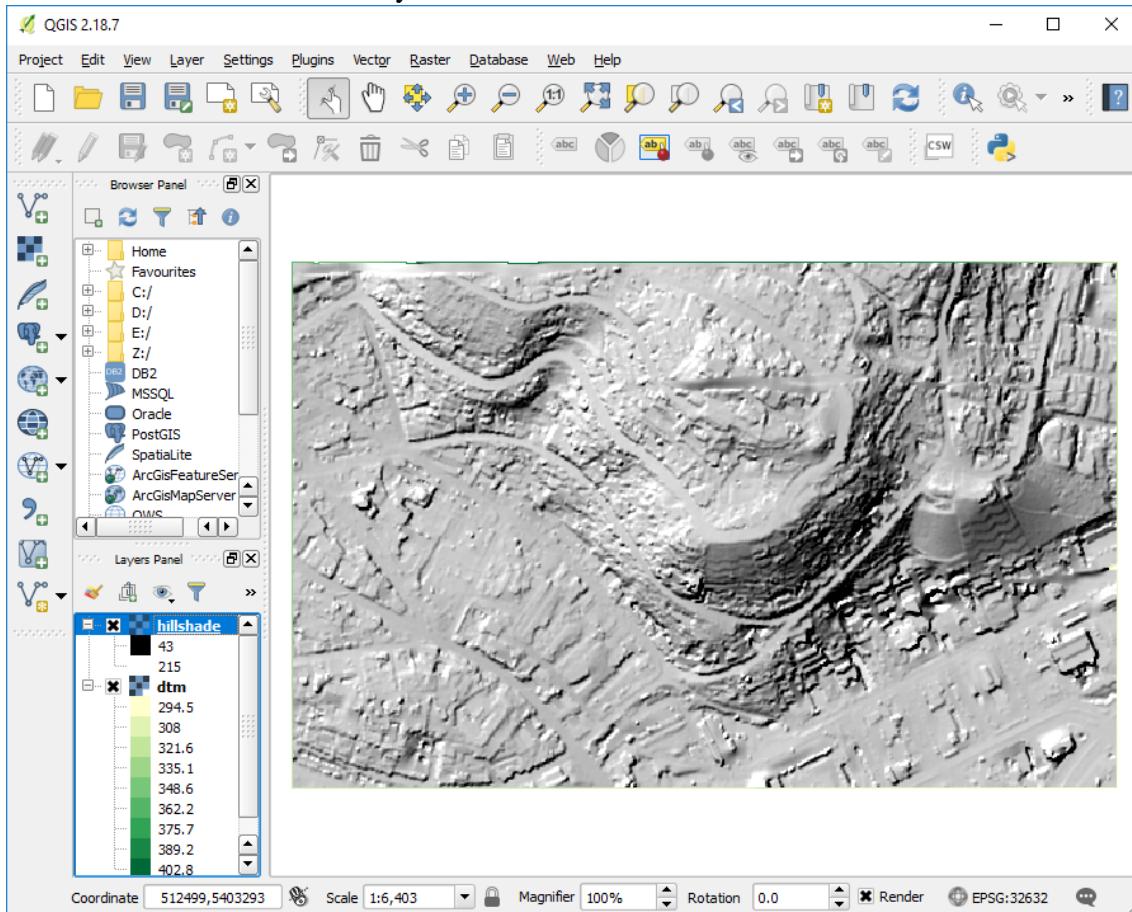


5. Click the window for the *Output file* and select a location to save the hillshade.tif



```
1 gdaldem hillshade ^
2     C:/Users/hobu/pdal/exercises/analysis/dtm/dtm.tif ^
3     C:/Users/hobu/pdal/exercises/analysis/dtm/hillshade.tif ^
4     -z 1.0 -s 1.0 -az 315.0 -alt 45.0 -of GTiff
```

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 81) can be used for large data, but it does not interpolate a typical TIN (https://en.wikipedia.org/wiki/Triangulated_irregular_network) surface model.

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 189) bindings and the filters.programmable and filters.predicate filters. 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 189) 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 31).

Note: Python allows you to enhance and build functionality that you can use in the context of other [Pipeline](#) (page 41) operations.

PDAL Pipeline

We're going to create a PDAL [Pipeline](#) (page 41) to tell PDAL to run our Python script in a filters.programmable stage.

```
1  {
2      "pipeline": [
3          {
4              "filename": "c:/Users/hobu/PDAL/exercies/
5              ↪python/athletic-fields.laz"
6          },
7          {
8              "type": "filters.programmable",
9              "function": "make_plot",
10             "module": "anything",
11             "pdalargs": {"filename": "histogram.png"}
12             ↪",
13             "script": "c:/Users/hobu/PDAL/exercies/python/
14             ↪histogram.py"
15         },
16     ]
17 }
```

```
14                     "type": "writers.null"
15                 }
16             ]
17 }
```

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 41) 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.

```
1 # import numpy
2 import numpy as np
3
4 # import matplotlib stuff and make sure to use the
5 # AGG renderer.
6 import matplotlib
7 matplotlib.use('Agg')
8 import matplotlib.pyplot as plt
9 import matplotlib.mlab as mlab
10
11 # This only works for Python 3. Use
12 # StringIO for Python 2.
13 from io import BytesIO
14
15 # The make_plot function will do all of our work. The
16 # filters.programmable filter expects a function name in the
17 # module that has at least two arguments -- "ins" which
18 # are numpy arrays for each dimension, and the "outs" which
19 # the script can alter/set/adjust to have them updated for
20 # further processing.
21 def make_plot(ins, outs):
```

```

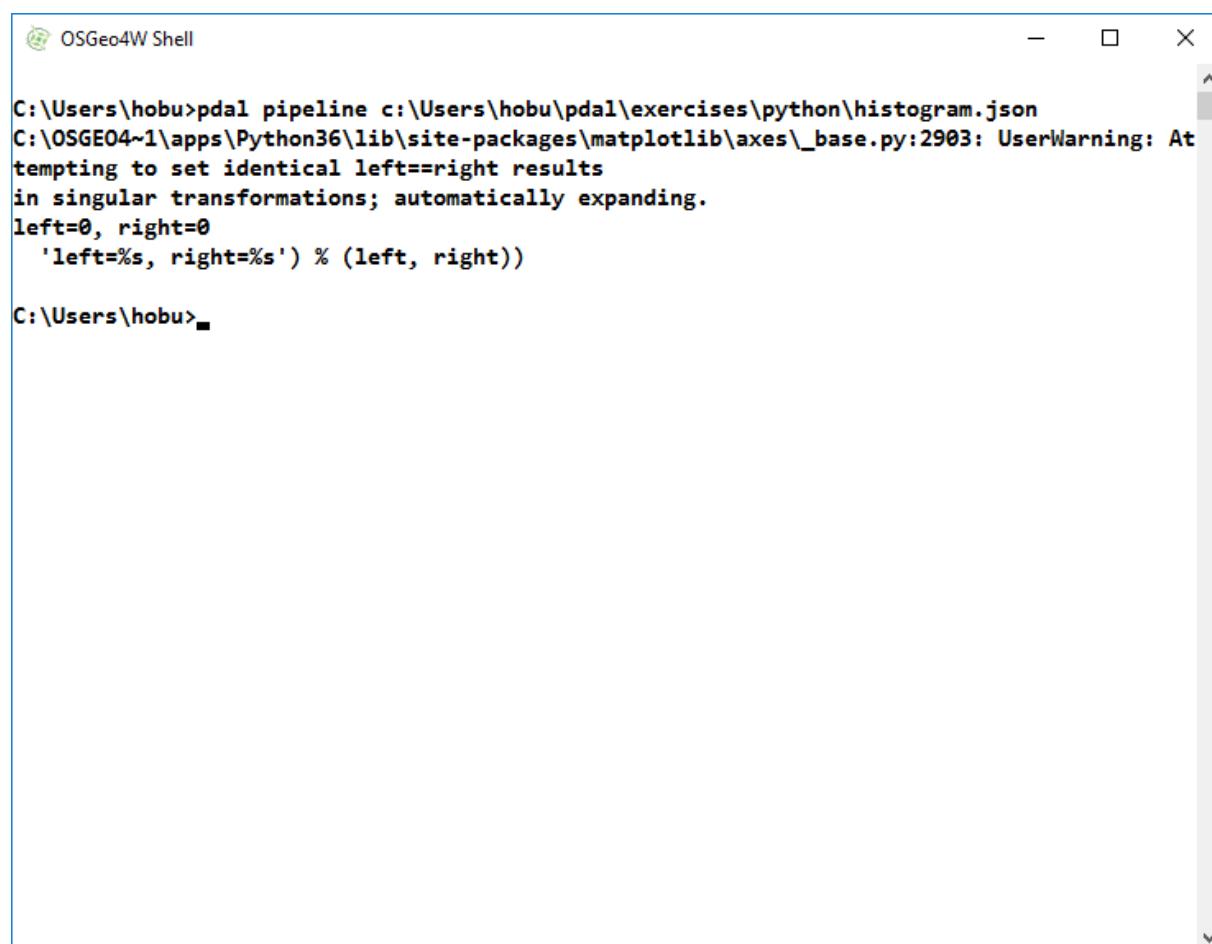
23 # figure position and row will increment
24 figure_position = 1
25 row = 1
26
27 fig = plt.figure(figure_position, figsize=(6, 8.5), dpi=300)
28
29 for key in ins:
30     dimension = ins[key]
31     ax = fig.add_subplot(len(ins.keys()), 1, row)
32
33     # histogram the current dimension with 30 bins
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:
68         filename = 'histogram.png'

```

```
68 # open up the filename and write out the
69 # bytes of the PNG stored in the BytesIO instance
70 o = open(filename, 'wb')
71 o.write(output.getvalue())
72 o.close()
73
74
75 # filters.programmable scripts need to
76 # return True to tell the filter it was successful.
77 return True
78
79
80
```

Run pdal pipeline

```
1 pdal pipeline c:/Users/hobu/PDAL/exercises/python/histogram.json
```

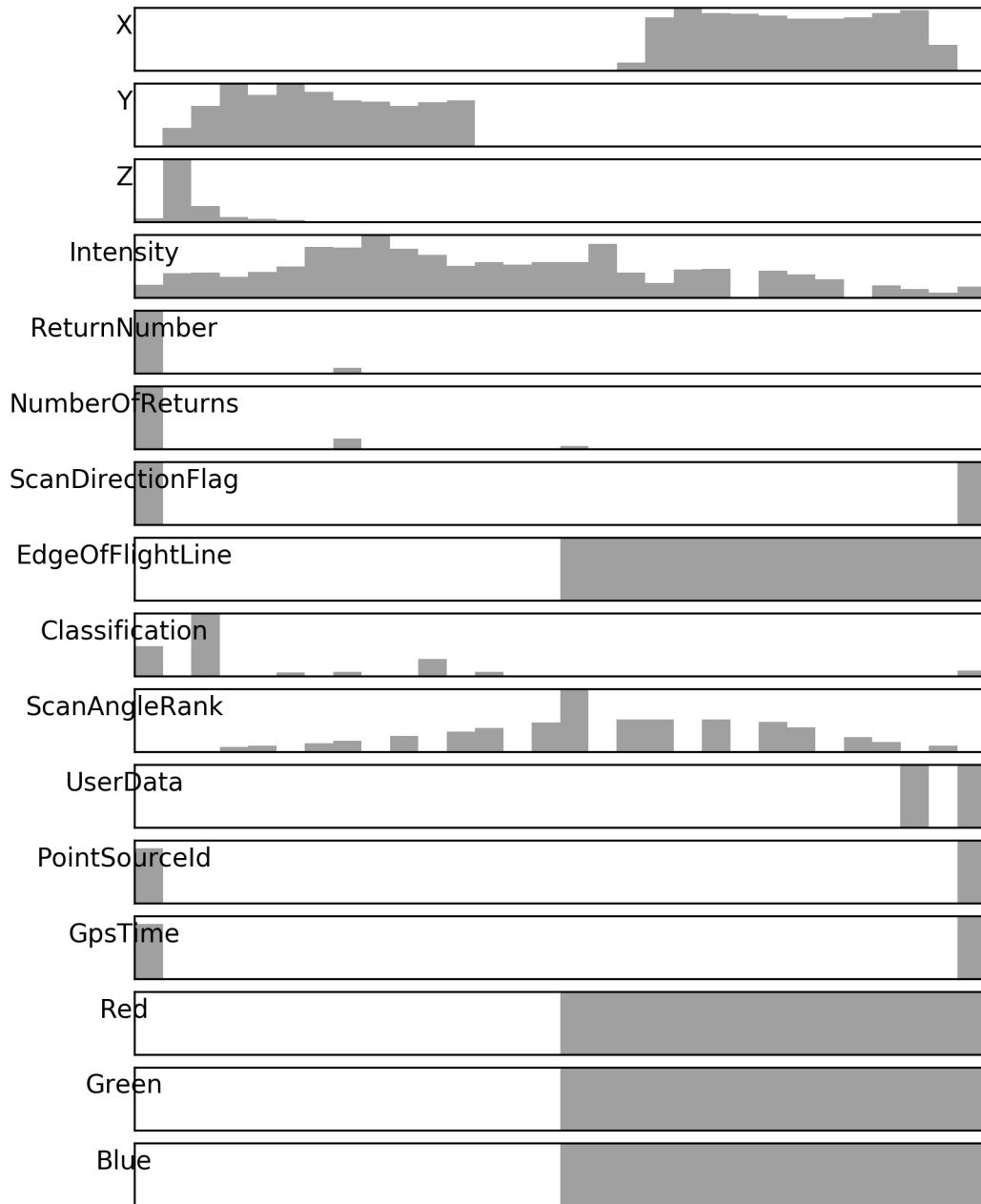


The screenshot shows a terminal window titled "OSGeo4W Shell". The command "pdal pipeline c:/Users/hobu/PDAL/exercises/python/histogram.json" is entered and executed. The output includes a UserWarning message from matplotlib about attempting to set identical left==right results in singular transformations; automatically expanding. The command then prints the values for left and right, which are "%s" and "%s" respectively, followed by a call to the print function. The prompt "C:\Users\hobu>" is visible at the bottom.

```
C:\Users\hobu>pdal pipeline c:/Users/hobu/PDAL/exercises/python/histogram.json
C:\OSGEO4~1\apps\Python36\lib\site-packages\matplotlib\axes\_base.py:2903: UserWarning: At
tempting to set identical left==right results
in singular transformations; automatically expanding.
left=0, right=0
'left=%s, right=%s') % (left, right))

C:\Users\hobu>
```


Output



Notes

1. `writers.null` (page 92) 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 242), 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.

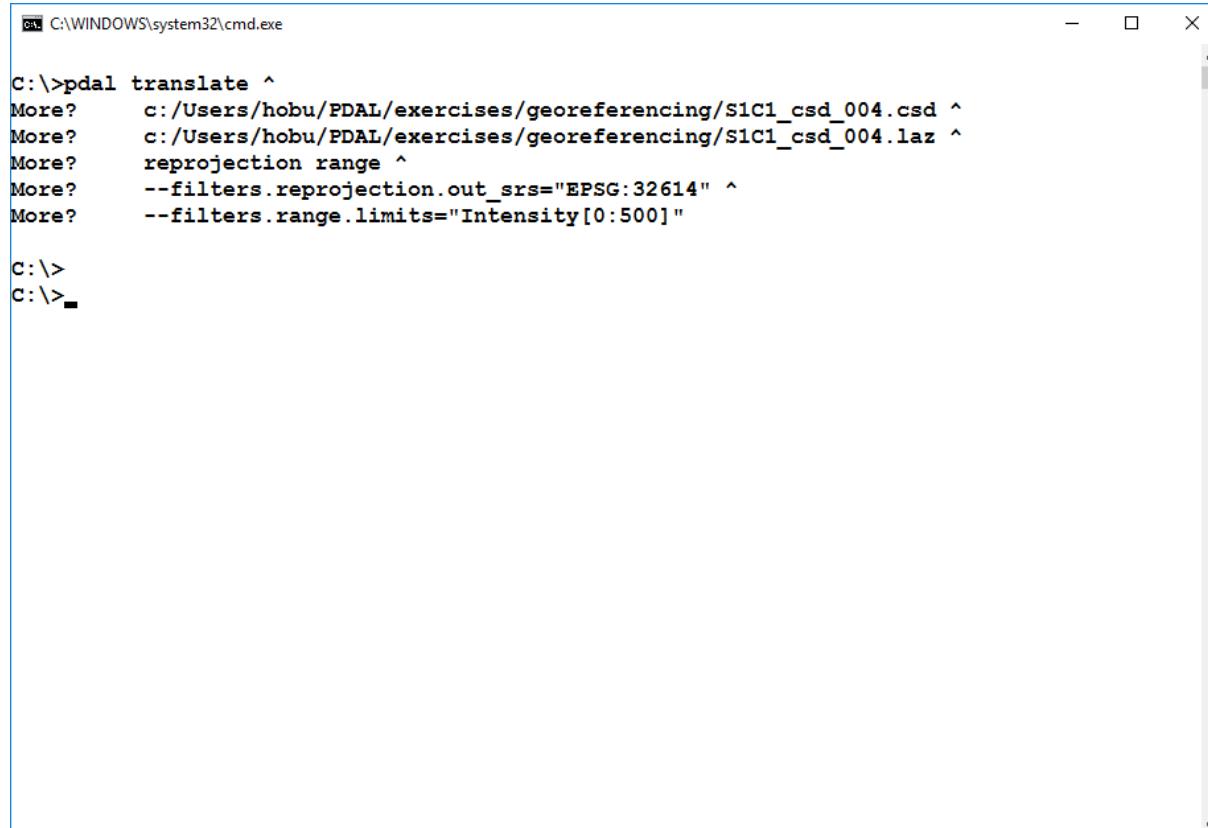
- Because these are raw data coming from the sensor, these data are noisy. In particular, there are a few points *very close* to the sensor which were probably caused by air returns or laser light reflecting off of part of the airplane or sensor. These points have very high intensity values, which will screw up our visualization. We will use the [filters.range](#) (page 170) 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 *OSGeo4W Shell*:

```
pdal translate ^
    c:/Users/hobu/PDAL/exercises/georeferencing/S1C1_csd_004.csd ^
    c:/Users/hobu/PDAL/exercises/georeferencing/S1C1_csd_004.laz ^
    reprojection range ^
    --filters.reprojection.out_srs="EPSG:32614" ^
    --filters.range.limits="Intensity[0:500]"
```



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is:

```
C:\>pdal translate ^
More?     c:/Users/hobu/PDAL/exercises/georeferencing/S1C1_csd_004.csd ^
More?     c:/Users/hobu/PDAL/exercises/georeferencing/S1C1_csd_004.laz ^
More?     reprojection range ^
More?     --filters.reprojection.out_srs="EPSG:32614" ^
More?     --filters.range.limits="Intensity[0:500]"
```

After the command is run, the prompt 'C:\>' appears twice, indicating the command has completed.

Visualization

View your georeferenced point cloud in <http://plas.io>.

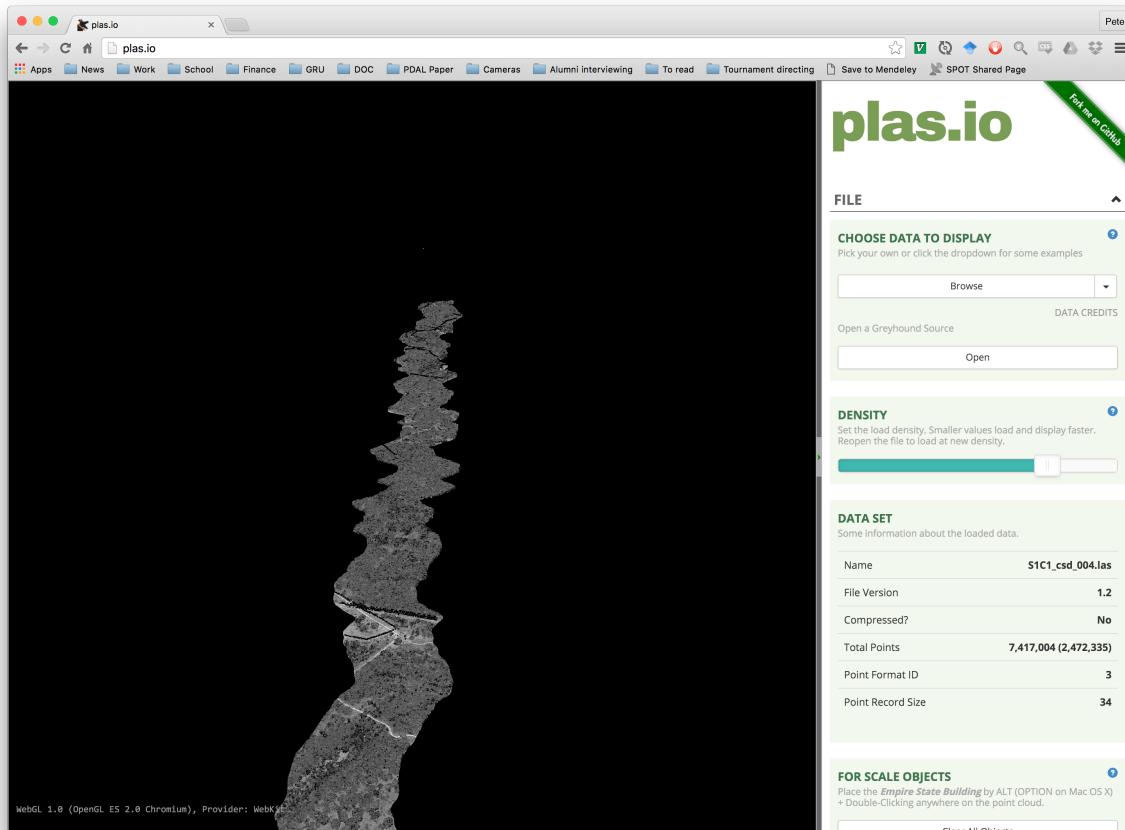


Fig. 11.7: Our airborne laser point cloud after georeferencing, reprojection, and intensity filtering.

11.1.5 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:

```
{
  "pipeline": [
    {
      "type": "read",
      "file": "S1C1_csd_004.las"
    }
  ]
}
```

```
        "type": "readers.greyhound",
        "url": "data.greyhound.io",
        "resource": "dublin",
        "threads": 8,
        "depth_end": 14,
        "bounds": "([-697041.0, -696241.0], [7045398.0, 7046086.
          ↪0], [-40, 400])"

    },
    {
        "type": "writers.las",
        "compression": "true",
        "minor_version": "2",
        "dataformat_id": "0",
        "filename": "st-stephens.laz"
    }
]
}
```

- Read data from a [Greyhound](#) (<http://greyhound.io/>) server using [*readers.greyhound*](#) (page 56) (See [*Greyhound*](#) (page 261))
- Thin it by 1.0 meter spacing using [*filters.sample*](#) (page 175) (See [*Thinning*](#) (page 287))
- Filter out noise using [*filters.outlier*](#) (page 151) (See [*Removing noise*](#) (page 278))
- Classify ground points using [*filters.smrf*](#) (page 176) (See [*Identifying ground*](#) (page 290))
- Compute height above ground using [*filters.hag*](#) (page 129)
- Generate a digital terrain model (DTM) using [*writers.gdal*](#) (page 81) (See [*Generating a DTM*](#) (page 299))
- Generate a average vegetative height model using [*writers.gdal*](#) (page 81)

Note: You should review specific [*Exercises*](#) (page 248) for specifics how to achieve each task.

11.1.6 Notes

Notes

Notes

Notes

Notes

Notes

Notes

CHAPTER
TWELVE

DEVELOPMENT

12.1 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 23) and users looking to use the PDAL API in their own applications should look [here](#) (page 386).

12.1.1 PDAL Architecture Overview

Author Andrew Bell

Contact andrew@hobu.co

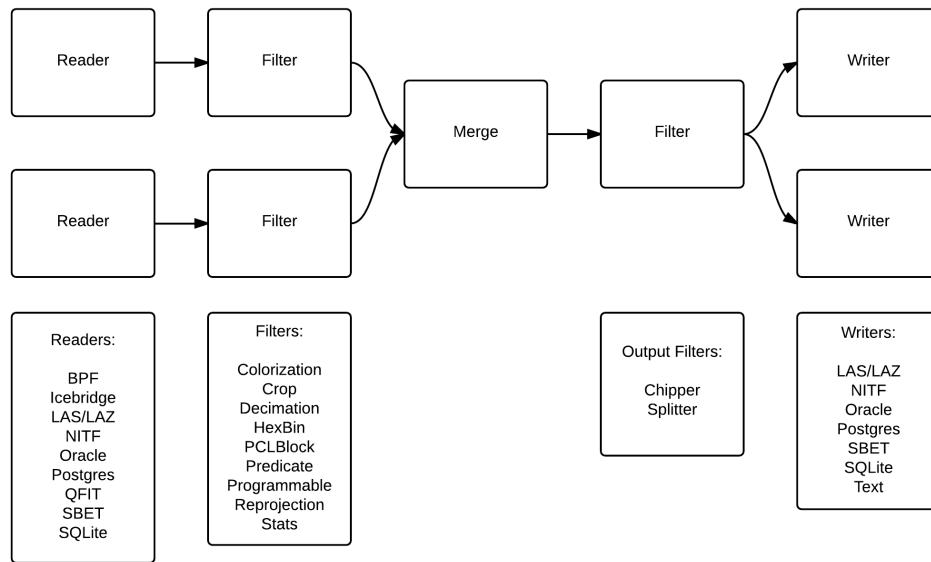
Date 5/15/2016

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 manipulations and can calculate some statistical, boundary and density data. 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 predefined 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 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 judgement call, but detection of errors earlier in the process allows faster termination of a command..

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 termination functions, such a 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 funciton, 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.

```
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.

```
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().

Readers should implement a function, read(), that will place the data from the input source into the provided point view:

```
point_count_t read(PointViewPtr view, point_count_t count)
```

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 a 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.

```

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--)
    {
        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);
        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;

    // 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:

```
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:

```
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);
        view->setField(Dimension::Id::Y, id, y);
        view->setField(Dimension::Id::Z, id, z);
    }
}
```

The filter simply loops through the points, retrieving the X, Y and Z values of each point, transforms those value 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:

```
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);  
}
```

12.1.2 Compilation

This section describes how to build and install PDAL under Windows, Linux, and Mac. PDAL's numerous *Dependencies* (page 349) can make it a challenge to build a fully-featured build.

See also:

Download (page 13) contains links to installable binaries for Windows, OSX, and RHEL Linux systems.

See also:

Install Docker (page 15) describes how to use Docker (<http://docker.io>) to get going with PDAL very quickly. This method is likely the fastest way to start using PDAL, especially if you just wish to use the *Applications* (page 23).

Contents:

Unix Compilation

Author Howard Butler

Contact howard@hobu.co

Date 10/27/2015

CMake (<http://www.cmake.org/>) 2.8.11+ is the prescribed tool for building from source, with CMake (<http://www.cmake.org/>) 3.0+ being desired. CMake (<http://www.cmake.org/>) is a cross-platform build system that provides a number of benefits, and its usage ensures a single, up-to-date build system for all PDAL-supported operating systems and compiler platforms.

Like a combination of autoconf/autotools, except that it works on Windows with minimal eye-stabbing pain, CMake (<http://www.cmake.org/>) is somewhat of a meta-building tool. It can be used to generate MSVC project files, GNU Makefiles, NMake files for MSVC, XCode projects on Mac OS X, and Eclipse projects (as well as many others). This functionality allows the PDAL project to avoid maintaining these build options by hand and target a single configuration and build platform.

This tutorial will describe how to build PDAL using CMake on a Unix platform. PDAL is known to compile on Linux 2.6's of various flavors and OSX with XCode.

See also:

Install Docker (page 15) contains an automated way to build PDAL and all of its dependencies. The [Dockerfile](https://github.com/PDAL/PDAL/blob/master/scripts/docker/Dockerfile) (<https://github.com/PDAL/PDAL/blob/master/scripts/docker/Dockerfile>) for PDAL build on Xenial is an excellent script describing how to build PDAL and all of its dependencies.

Note: [*Dependencies*](#) (page 349) contains more information about specific library version requirements and notes about building or acquiring them.

Using “Unix Makefiles” on Linux

Get the source code

See [*Development Source*](#) (page 14) for how to obtain the latest development version or visit [*Download*](#) (page 13) to get the latest released version.

Prepare a build directory

CMake allows you to generate different builders for a project, and in this example, we are going to generate a “Unix Makefiles” builder for PDAL on Mac OS X.

```
$ cd PDAL
$ mkdir makefiles
$ cd makefiles
```

Configure base library

Configure the basic core library for the “Unix Makefiles” target:

```
$ cmake -G "Unix Makefiles" ../
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Checking whether C compiler has -isysroot
-- Checking whether C compiler has -isysroot - yes
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Checking whether CXX compiler has -isysroot
-- Checking whether CXX compiler has -isysroot - yes
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
```

```
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Enable PDAL utilities to build - done
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/hobu/dev/git/PDAL-cmake/
→makefiles
```

Note: The `./cmake/examples/hobu-config.sh` shell script contains a number of common settings that I use to configure my *Homebrew*-based Macintosh system.

Issue the `make` command

This will build a base build of the library, with no extra libraries being configured.

Run `make install` and test your installation with a `pdal_test` command

`make install` will install the *utilities* (page 23) in the location that was specified for ‘`CMAKE_INSTALL_PREFIX`’. Once installed, ensure that you can run `pdal info`.

Configure your Optional Libraries.

By checking the “on” button for each, CMake may find your installations of these libraries, but in case it does not, set the following variables, substituting accordingly, to values that match your system layout.

GDAL (http://www.gdal.org)	GDAL_CONFIG	/usr/local/bin/gdal-config
	GDAL_INCLUDE_DIR	/usr/local/include
	GDAL_LIBRARY	/usr/local/lib/libgdal.so
GeoTIFF (http://trac.osgeo.org/geotiff)	GEO-TIFF_INCLUDE_DIR	/usr/local/include
	GEOTIFF_LIBRARY	/usr/local/lib/libgeotiff.so
OCI (http://www.oracle.com/technology/tech/oci/oralex.html)	ORA-CLE_INCLUDE_DIR	/home/oracle/sdk/include
	ORA-CLE_NNZ_LIBRARY	/home/oracle/libnnz10.so
	ORA-CLE_OCCI_LIBRARY	/home/oracle/libocci.so
	ORA-CLE_OCIEI_LIBRARY	/home/oracle/libociei.so
	ORA-CLE_OCI_LIBRARY	/home/oracle/libclntsh.so

CCMake and cmake-gui

Warning: The following was just swiped from the libLAS compilation document and it has not been updated for PDAL. The basics should be the same, however. Please ask on the [mailing list](#) (page 39) if you run into any issues.

While [CMake](http://www.cmake.org/) (<http://www.cmake.org/>) can be run from the command-line, and this is the preferred way for many individuals, it can be much easier to run CMake from a GUI. Now that we have a basic library building, we will use CMake's GUIs to help us configure the rest of the optional components of the library. Run `ccmake ..` for the [Curses](#) (http://en.wikipedia.org/wiki/Curses_%28programming_library%29) interface or `cmake-gui ..` for a GUI version.

Note: If your arrow keys are not working with in CCMake, use CTRL-N and CTRL-P to move back and forth between the options.

Build and install

Once you have configured your additional libraries, you can install the software. The main pieces that will be installed are:

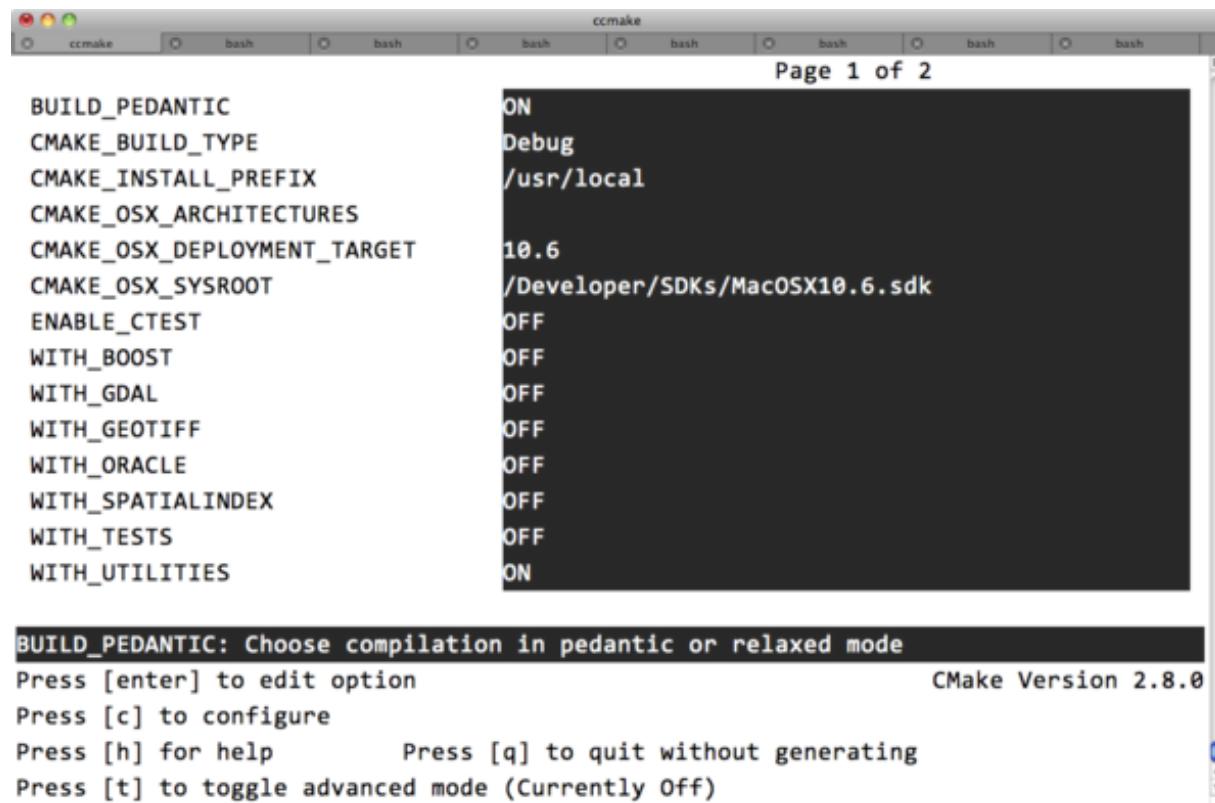


Fig. 12.1: Running the [Curses](http://en.wikipedia.org/wiki/Curses_%28programming_library%29) (http://en.wikipedia.org/wiki/Curses_%28programming_library%29) [CMake](http://www.cmake.org/) (<http://www.cmake.org/>) interface. This interface is available to all unix-like operating systems.

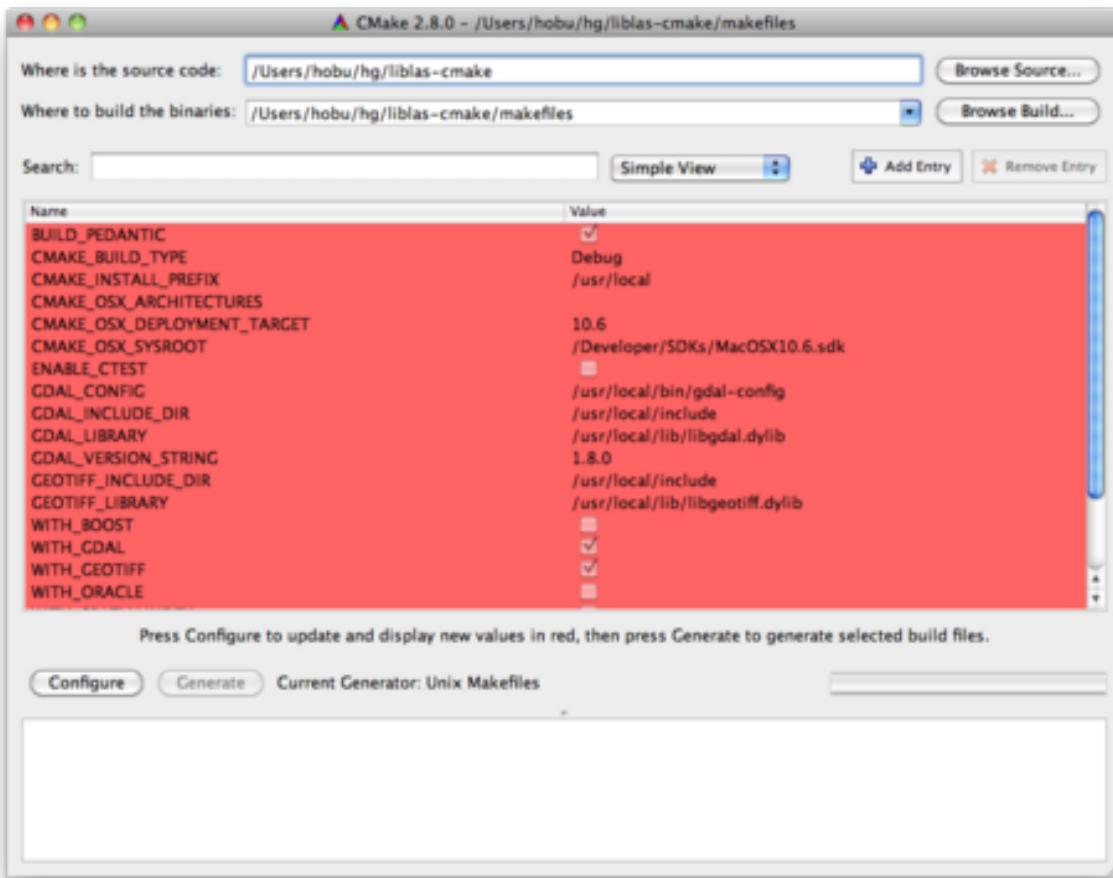


Fig. 12.2: Running the `cmake-gui` CMake (<http://www.cmake.org/>) interface. This interface is available on Linux, Windows, and Mac OS X.

- PDAL headers (typically in a location `./include/pdal/...`)
- PDAL C++ (`PDAL.a` or `PDAL.so`) library
- PDAL C (`PDAL_c.a` or `PDAL_c.so`) library
- *Utility* (page 23) programs

```
make install
```

Using “XCode” on OS X

Get the source code

See [Development Source](#) (page 14) for how to obtain the latest development version or visit [Download](#) (page 13) to get the latest released version.

Prepare a build directory

CMake allows you to generate different builders for a project, and in this example, we are going to generate an “Xcode” builder for PDAL on Mac OS X. Additionally, we’re going to use an alternative compiler – [LLVM](#) (<http://llvm.org/>) – which under certain situations can produce much faster code on Mac OS X.

```
$ export CC=/usr/bin/llvm-gcc
$ export CXX=/usr/bin/llvm-g++
$ cd PDAL
$ mkdir xcode
$ cd xcode/
```

Configure base library

Configure the basic core library for the Xcode build:

```
$ cmake -G "Xcode" ..
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Checking whether C compiler has -isysroot
-- Checking whether C compiler has -isysroot - yes
-- Check for working C compiler: /usr/bin/llvm-gcc
-- Check for working C compiler: /usr/bin/llvm-gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
```

```
-- Checking whether CXX compiler has -isysroot
-- Checking whether CXX compiler has -isysroot - yes
-- Check for working CXX compiler: /usr/bin/llvm-g++
-- Check for working CXX compiler: /usr/bin/llvm-g++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Enable PDAL utilities to build - done
-- Enable PDAL unit tests to build - done
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/hobu/hg/PDAL-cmake/xcode
```

Alternatively, if you have [KynGChaos](http://www.kyngchaos.com/software/unixport) (<http://www.kyngchaos.com/software/unixport>) frameworks for [GDAL](http://www.gdal.org) (<http://www.gdal.org>) and [GeoTIFF](http://trac.osgeo.org/geotiff) (<http://trac.osgeo.org/geotiff>) installed, you can provide locations for those as part of your `cmake` invocation:

```
$ cmake -G "Xcode" \
-D GDAL_CONFIG=/Library/Frameworks/GDAL.framework/Programs/gdal-
config \
-D GEOTIFF_INCLUDE_DIR=/Library/Frameworks/UnixImageIO.framework/
unix/include \
-D GEOTIFF_LIBRARY=/Library/Frameworks/UnixImageIO.framework/unix/
lib/libgeotiff.dylib \
..
```

Note: I recommend that you use in [Homebrew](http://brew.sh/) (<http://brew.sh/>) for [GDAL](http://www.gdal.org) (<http://www.gdal.org>) and friends. Its configuration is featureful and up-to-date.

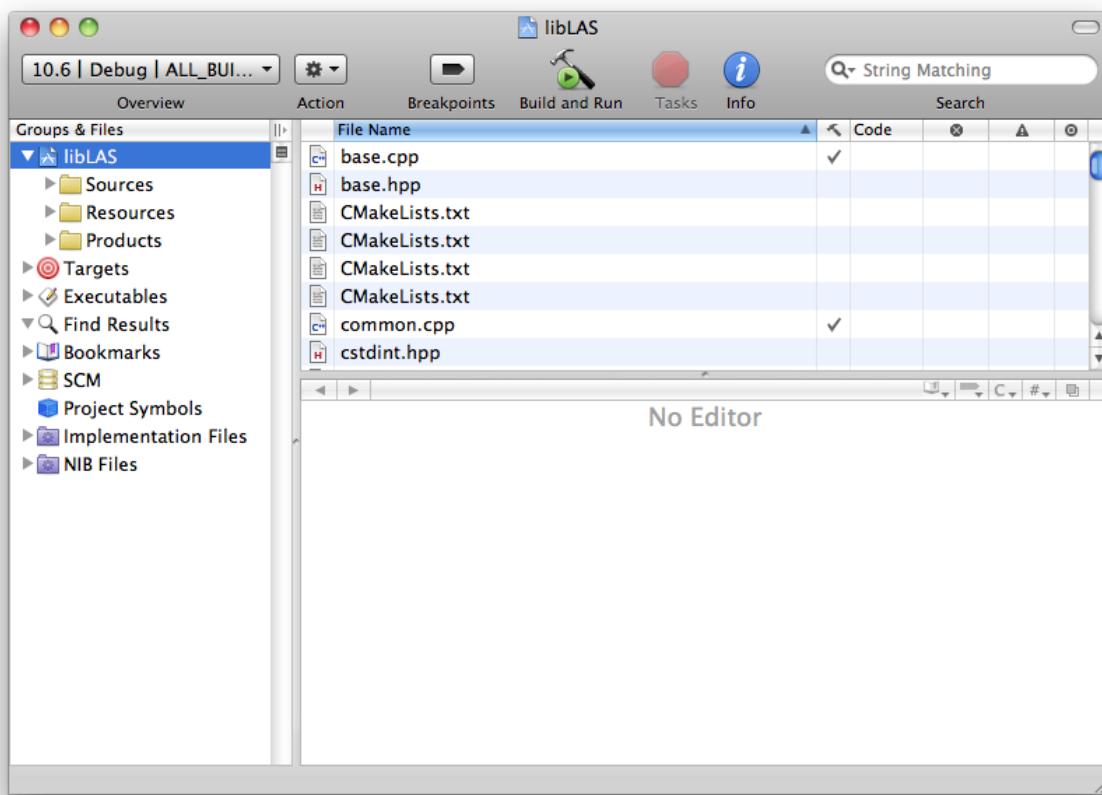
```
$ open PDAL.xcodeproj/
```

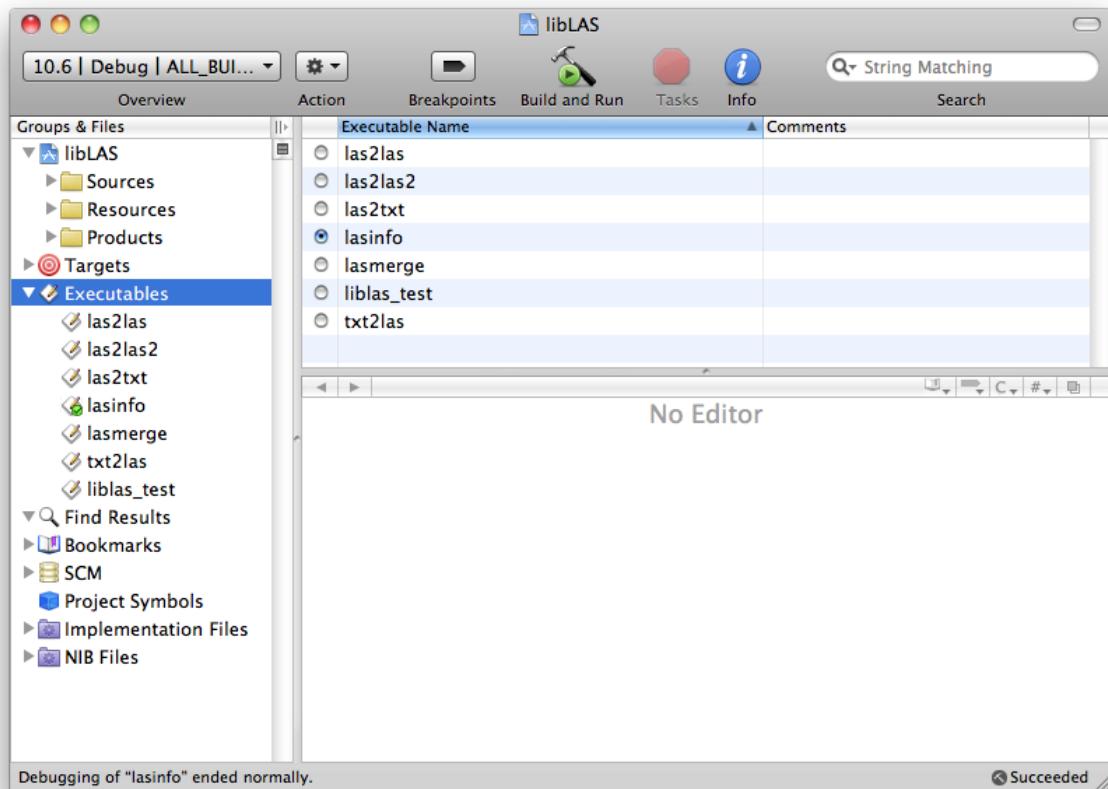
Set default command for XCode

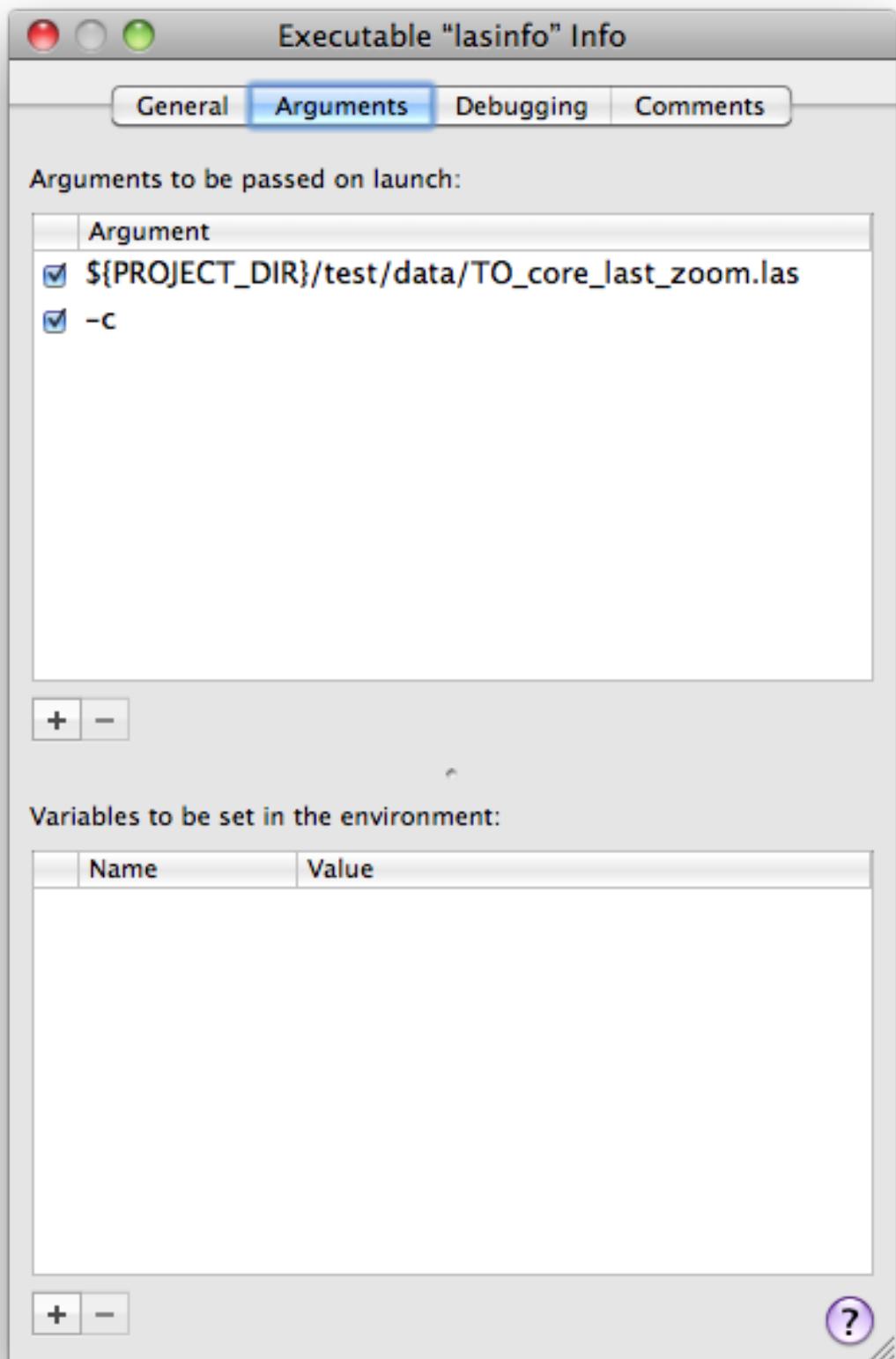
Set the default executable for the project to be `lasinfo` by opening the “Executables” tree, choosing “`lasinfo`,” and clicking the bubble next to the “Executable name” in the right-hand panel.

Set arguments for `pdal_test`

Set the arguments for `pdal_test` so it can be run from within XCode. We use the `${PROJECT_DIR}` environment variable to be able to tell `pdal_test` the location of our test file. This is similar to the [*same command*](#) (page 336) above in the “Unix Makefiles” section.







Configure Optional Libraries

As *before* (page 336), use `ccmake . . /` or `cmake-gui . . /` to configure your *Dependencies* (page 349).

Building Under Windows

Author Howard Butler

Contact howard@hobu.co

Author Michael Rosen

Contact unknown at lizardtech.com

Date 11/02/2017

Note: [OSGeo4W](https://trac.osgeo.org/osgeo4w/) (<https://trac.osgeo.org/osgeo4w/>) 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.

See also:

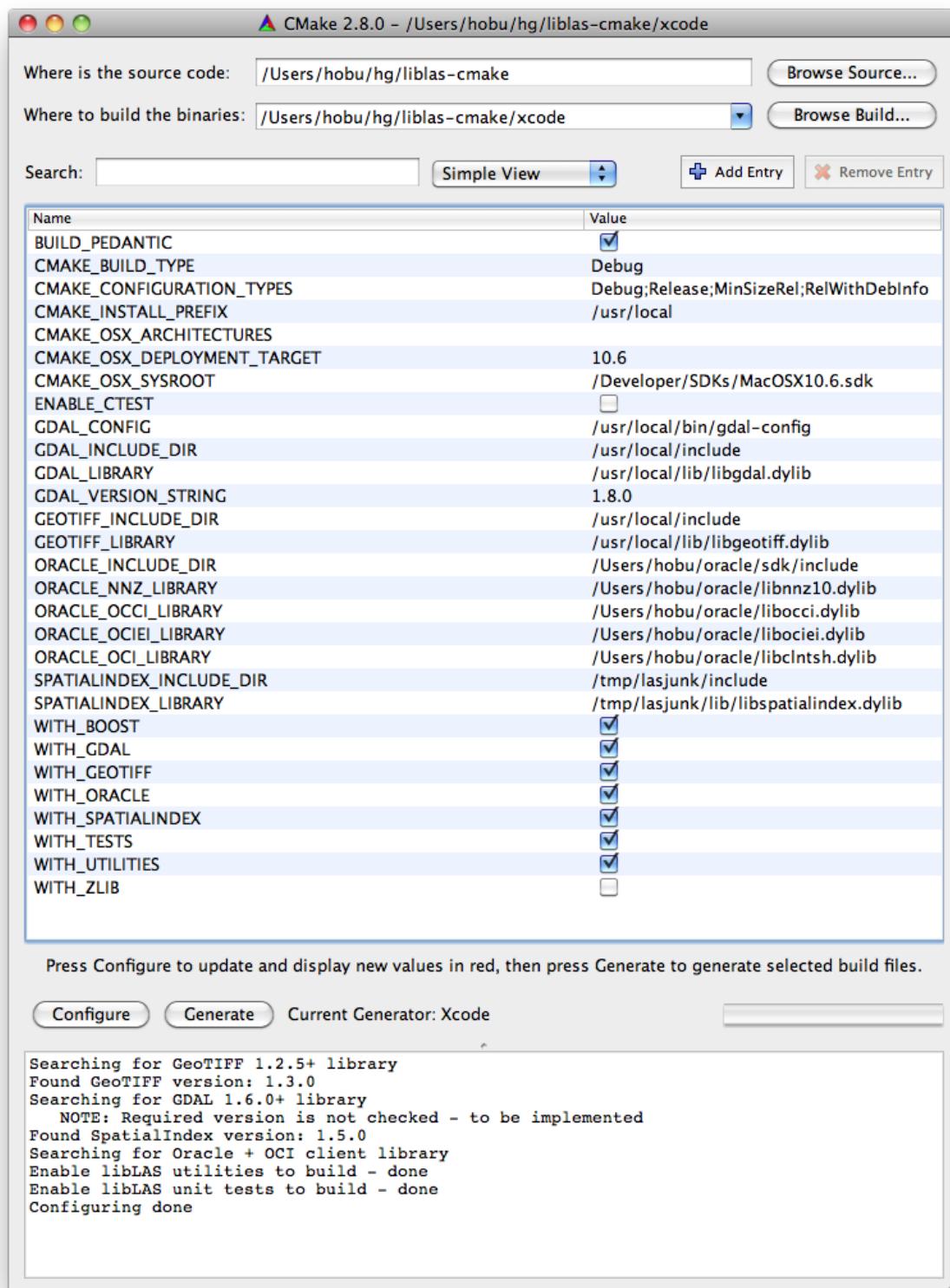
[Install Docker](#) (page 15) describes a way to get a PDAL build and all of its dependencies. If you just want to apply PDAL commandline operations to data, this mechanism is likely to be much faster than compiling your own.

Introduction

Pre-built binary packages for Windows are available via [OSGeo4W](#) (<https://trac.osgeo.org/osgeo4w/>) (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 OSGeo4W as your base, and anything more advanced is beyond the scope of the document.

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 OSGeo4W 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.

The AppVeyor configuration installs OSGeo4W and all of PDAL's prerequisites via the command line.

After downloading the OSGeo4W setup (http://download.osgeo.org/osgeo4w/osgeo4w-setup-x86_64.exe), you can invoke it via the command line to install PDAL's prerequisite packages.

```
C:\temp\osgeo4w-setup.exe -q -k -r -A -s http://download.osgeo.org/
  ↳osgeo4w/ -a x86_64 ^
    -P eigen,gdal,geos,hexer,iconv,laszip,libgeotiff,libpq,libtiff,
  ↲^
    libxml2,msys,nitro,laz-perf,proj,zlib,python3-core,python3-
  ↳devel, ^
    python3-numpy,oci,oci-devel,laz-perf,jsoncpp -R c:/OSGeo4W64
```

Note: The package list here might change over time. The canonical location to learn the OSGeo4W prerequisite list for PDAL is the appveyor.yml file in PDAL's source tree.

See also:

If you don't wish to run via the command line, you can choose the GUI for installation. Visit [workshop-osgeo4w](#) for a description, and then choose all of the listed support libraries (minus PDAL of course) to schedule them for installation.

Warning: There are a number of package scripts that assume c:/OSGeo4W64 as the installation path, and it is likely that you will run into some trouble attempting to install in other locations. It's possible it will work with some elbow grease, but it might not work out of the box.

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 1.6-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/cmake/examples/hobu-windows.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 need to have your OSGeo4W shell active to enable access to PDAL’s dependencies. Issue c:\osgeo4w64\bin\o4w_env.bat in your shell to activate it.

Dependencies

Author Howard Butler

Contact howard@hobu.co

Date 11/03/2015

PDAL explicitly stands on the shoulders of giants that have come before it. Specifically, PDAL depends on a number of libraries to do its work. Most are not required. For optional dependencies, PDAL utilizes a dynamically-linked plugin architecture that loads them at runtime.

Required Dependencies

GDAL

PDAL uses GDAL for spatial reference system description manipulation, and image reading supporting for the NITF driver, and [writers.oci](#) (page 93) support. In conjunction with [GeoTIFF](#) (<http://trac.osgeo.org/geotiff>), GDAL is used to convert GeoTIFF keys and OGC WKT SRS description strings into formats required by specific drivers. While PDAL can be built without GDAL support, if you want SRS manipulation and description ability, you must have GDAL (and [GeoTIFF](#) (<http://trac.osgeo.org/geotiff>)) linked in at compile time.

Obtain [GDAL](#) (<http://www.gdal.org>) via whatever method is convenient. Linux platforms such as [Debian](#) (<http://www.debian.org>) have [DebianGIS](#) (<http://wiki.debian.org/DebianGis>), Mac OS X has the [KyngChaos](#) (<http://www.kyngchaos.com/software/unixport>) software frameworks, and Windows has the [OSGeo4W](#) (<http://trac.osgeo.org/osgeo4w/>) platform.

- GDAL 1.9+ is required.

Warning: If you are using [OSGeo4W](http://trac.osgeo.org/osgeo4w/) (<http://trac.osgeo.org/osgeo4w/>) as your provider of GDAL, you must make sure to use the GDAL 1.9 package.

GeoTIFF

PDAL uses GeoTIFF in conjunction with GDAL for GeoTIFF key support in the LAS driver. Obtain [GeoTIFF](http://trac.osgeo.org/geotiff) (<http://trac.osgeo.org/geotiff>) from the same place you got [GDAL](http://www.gdal.org) (<http://www.gdal.org>).

- libgeotiff 1.3.0+ is required

Note: GDAL surreptitiously embeds a copy of [GeoTIFF](http://trac.osgeo.org/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) (<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. [OSGeo4W](http://trac.osgeo.org/osgeo4w/) (<http://trac.osgeo.org/osgeo4w/>) users, including those using that platform to link and build PDAL themselves, do not need to worry about this issue.

Proj.4

[Proj.4](http://trac.osgeo.org/proj) (<http://trac.osgeo.org/proj>) is the projection engine that PDAL uses for the [*filters.reprojection*](#) (page 173) filter. It is used by GDAL.

Note: Proj.4 4.9.0+ is required if you need vertical datum transformation support. Otherwise, older versions should be sufficient.

Optional Dependencies

libxml2

libxml2 (<http://xmlsoft.org>) is used to serialize PDAL dimension descriptions into XML for the database drivers such as [*writers.oci*](#) (page 93), [*readers.sqlite*](#) (page 75), or [*readers.pgpointcloud*](#) (page 68)

Note: libxml 2.7.0+ is required. Older versions may also work but are untested.

OCI (<http://www.oracle.com/technology/tech/oci/index.html>)

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 CMake configuration can find your install. OCI is used by both *writers.oci* (page 93) and *readers.oci* (page 66) for Oracle Point Cloud read/write support.

Warning: OCI (<http://www.oracle.com/technology/tech/oci/index.html>)’s libraries are inconsistently named. You may need to create symbolic links for some library names in order for the [CMake](#) (<http://www.cmake.org>) to find them:

```
cd $ORACLE_HOME
ln -s libocci.so.11.1 libocci.so
ln -s libclntsh.so.11.1 libclntsh.so
ln -s libociei.so.11.1 libociei.so
```

- OCI 10g+ is required.

Note: MSVC should only require the oci.lib and oci.dll library and dlls.

Hexer

[Hexer](#) (page 351) is a library with a simple *CMake*-based build system that provides simple hexagon gridding of large point sets for density surface generation and boundary approximation. It can be obtained via [github.com at https://github.com/hobu/hexer](https://github.com/hobu/hexer) It is used by *filters.hexbin* (page 134) to output density surfaces and boundary approximations.

Nitro

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 90). PDAL can only use a fork of Nitro located at <http://github.com/hobu/nitro> instead of the mainline tree for two reasons:

1. The fork contains a simple *CMake*-based build system
2. The fork properly dynamically links on Windows to maintain LGPL compliance.

It is expected that the fork will go away once these items are incorporated into the main source tree.

LASzip

[LASzip](http://laszip.org) (<http://laszip.org>) is a library with a simple *CMake*-based build system that provides periodic compression of [ASPRS LAS](http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html) (<http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html>) data. It is used by the *writers.las* (page 85) and *readers.las* (page 59) to provide compressed LAS support.

laz-perf

In addition to [LASzip](http://laszip.org) (<http://laszip.org>), you can use the alternative [laz-perf](https://github.com/verma/laz-perf/) (<https://github.com/verma/laz-perf/>) library. [laz-perf](https://github.com/verma/laz-perf/) (<https://github.com/verma/laz-perf/>) provides slightly faster decompression capability for typical LAS files. It is also used as a compression type for *writers.oci* (page 93) and *writers.sqlite* (page 101)

PCL

The [Point Cloud Library \(PCL\)](http://pointclouds.org) (<http://pointclouds.org>) is used by the *pcl* (page 31), *writers.pcd* (page 97), *readers.pcd* (page 67), and *filters.pclblock* (page 156) to provide support for various PCL-related operations.

PCL must be 1.7.2+. We do our best to keep this up-to-date with PCL master.

Note: [Homebrew](http://brew.sh) (<http://brew.sh>)-based OSX builds use PCL 1.7.2, but you may need to switch off [VTK](http://vtk.org) (<http://vtk.org>) support depending on the configuration.

12.1.3 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.

12.1.4 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 31) and [pdal translate](#) (page 36) 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 such by calling addEncoded(). Programmers can specify the type of a node explicitly by calling addWithType(). 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:

```
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("nulluidnode", 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:

```
{
  "writers.test": {
    "intlist": [
      45,
      55
    ],
    "nodename": "Some string data",
    "parent": {
      "nulluidnode": "00000000-0000-0000-0000-000000000000",
      "num_in_list":
```

```
[  
    66  
]  
}  
}  
}
```

12.1.5 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 reader stage.

Note: If you implement your own *Readers* (page 50) that conforms to PDAL's *pdal::Stage* (page 420), you can implement a simple read-filter-write pipeline using *Pipeline* (page 41) and not have to code anything explicit yourself.

Includes

First, our code.

```
#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);
    table.layout()->registerDim(Dimension::Id::Y);
    table.layout()->registerDim(Dimension::Id::Z);

    PointViewPtr view(new PointView(table));

    fillView(view);

    BufferedReader 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.

```
#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.

```
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 BufferedReader and sends it to a writer.

```
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 334) 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_minimum_required(VERSION 3.6)
project(WritingTutorial)

find_package(PDAL 1.6.0 REQUIRED CONFIG)

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 384) for an explanation of the basic CMakeLists.

Begin by configuring your project using CMake (shown here on Unix) and building using make.

```
$ cd /PATH/TO/WRITING/TUTORIAL  
$ mkdir build  
$ cd build  
$ cmake ..  
$ make
```

After the project is built, you can run it by typing:

```
$ ./tutorial
```

12.1.6 Writing a filter

Author Bradley Chambers

Contact brad.chambers@gmail.com

Date 11/02/2017

PDAL can be extended through the development of filter functions.

See also:

For more on filters and their role in PDAL, please refer to [PDAL Architecture Overview](#) (page 323).

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 125), 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.pmf](#) (page 159), 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.

```
1 // MyFilter.hpp
2
3 #pragma once
4
5 #include <pdal/Filter.hpp>
6 #include <pdal/Stage.hpp>
7
8 #include <memory>
9
10 namespace pdal
11 {
12
13     class Options;
14     class PointLayout;
15     class PointView;
16
17     class PDAL_DLL MyFilter : public Filter
18     {
19     public:
20         MyFilter() : Filter()
21         { }
22
23         static void * create();
24         static int32_t destroy(void *);
25         std::string getName() const;
26
27     private:
28         double m_value;
29         Dimension::Id m_myDimension;
30
31         virtual void addDimensions(PointLayoutPtr layout);
32         virtual void addArgs(ProgramArgs& args);
33         virtual PointViewSet run(PointViewPtr view);
34
35         MyFilter& operator=(const MyFilter&); // not implemented
36         MyFilter(const MyFilter&); // not implemented
37     };
38
39 } // namespace pdal
```

This header should be relatively straightforward, but we will point out three methods that must be declared for the plugin interface to be satisfied.

```

static void * create();
static int32_t destroy(void *);
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.

```

1 // MyFilter.cpp
2
3 #include "MyFilter.hpp"
4
5 #include <pdal/Options.hpp>
6 #include <pdal/pdal_macros.hpp>
7 #include <pdal/PointTable.hpp>
8 #include <pdal/PointView.hpp>
9 #include <pdal/StageFactory.hpp>
10 #include <pdal/util/ProgramArgs.hpp>
11
12 namespace pdal
13 {
14
15     static PluginInfo const s_info =
16         PluginInfo("filters.name", "My awesome filter",
17                     "http://link/to/documentation");
18
19 CREATE_STATIC_PLUGIN(1, 0, MyFilter, Filter, s_info)
20
21 std::string MyFilter::getName() const
22 {
23     return s_info.name;
24 }
25
26 void MyFilter::addArgs(ProgramArgs& args)
27 {
28     args.add("param", "Some parameter", m_value, 1.0);
29 }
30
31 void MyFilter::addDimensions(PointLayoutPtr layout)
32 {
33     layout->registerDim(Dimension::Id::Intensity);
34     m_myDimension = layout->registerOrAssignDim("MyDimension",

```

```
35         Dimension::Type::Unsigned8);
36     }
37
38     PointViewSet MyFilter::run(PointViewPtr input)
39     {
40         PointViewSet viewSet;
41         viewSet.insert(input);
42         return viewSet;
43     }
44
45 } // 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 the macros in `pdal_macros.hpp` to do just this.

We begin by creating a `PluginInfo` struct containing three identifying elements - the filter name, description, and a link to documentation.

```
1 static PluginInfo const s_info =
2     PluginInfo("filters.name", "My awesome filter",
3                 "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`).

Next, we pass the following to the `CREATE_STATIC_PLUGIN` macro, in order: PDAL plugin ABI major version, PDAL plugin ABI minor version, filter class name, stage type (`Filter`), and our `PluginInfo` struct.

```
CREATE_STATIC_PLUGIN(1, 0, MyFilter, Filter, s_info)
```

To create a shared plugin, we simply change `CREATE_STATIC_PLUGIN` to `CREATE_SHARED_PLUGIN`.

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.

```
1 std::string MyFilter::getName() const
2 {
3     return s_info.name;
4 }
```

Now that the filter has implemented the proper plugin interface, we will begin to implement some methods that actually implement the filter. First, `getDefaults()` is used to advertise those options that the filter provides. Within PDAL, this is primarily used as a means of displaying options via the PDAL CLI with the `--options` argument. It provides the user with the option names, descriptions, and default values.

```

1 void MyFilter::addArgs(ProgramArgs& args)
2 {
3     args.add("param", "Some parameter", m_value, 1.0);
4 }
```

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`.

```

1 void MyFilter::addDimensions(PointLayoutPtr layout)
2 {
3     layout->registerDim(Dimension::Id::Intensity);
4     m_myDimension = layout->registerOrAssignDim("MyDimension",
5                                                 Dimension::Type::Unsigned8);
6 }
```

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()`.

```

1 PointViewSet MyFilter::run(PointViewPtr input)
2 {
3     PointViewSet viewSet;
4     viewSet.insert(input);
5     return viewSet;
6 }
```

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!

StageFactory

As of this writing, users must also make a couple of changes to `StageFactory.cpp` to properly register static plugins only (this is not required for shared plugins). It is our goal to eventually remove this requirement to further streamline development of add-on plugins.

Note: Modification of `StageFactory` is required for STATIC plugins only. Dynamic plugins are registered at runtime.

First, add the following line to the beginning of `StageFactory.cpp` (adjusting the path and filename as necessary).

```
#include <MyFilter.hpp>
```

Next, add the following line of code to the `StageFactory` constructor.

```
PluginManager::initializePlugin(MyFilter_InitPlugin);
```

Compilation

Set up a `CMakeLists.txt` file to compile your filter against PDAL:

```
1 cmake_minimum_required(VERSION 2.8.12)
2 project(FilterTutorial)
3
4 find_package(PDAL 1.6.0 REQUIRED CONFIG)
5
6 add_library(pdal_plugin_filter_myfilter SHARED MyFilter.cpp)
7 target_link_libraries(pdal_plugin_filter_myfilter PRIVATE ${PDAL_
    ↴LIBRARIES})
8 target_include_directories(pdal_plugin_filter_myfilter PRIVATE
    ${PDAL_INCLUDE_DIRS})
```

12.1.7 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.

```
1 // MyKernel.hpp
2
3 #pragma once
4
5 #include <pdal/Kernel.hpp>
6 #include <pdal/plugin.hpp>
7
8 #include <string>
```

```

9
10 namespace pdal
11 {
12
13     class PDAL_DLL MyKernel : public Kernel
14     {
15         public:
16             static void * create();
17             static int32_t destroy(void *);
18             std::string getName() const;
19             int execute(); // override
20
21         private:
22             MyKernel();
23             void addSwitches(ProgramArgs& args);
24
25             std::string m_input_file;
26             std::string m_output_file;
27     };
28
29 } // namespace pdal

```

As with other plugins, the MyKernel class needs to have the following three methods declared for the plugin interface to be satisfied:

```

static void * create();
static int32_t destroy(void *);
std::string getName() const;

```

The source

Again, we start with a full listing of the kernel source.

```

1 // MyKernel.cpp
2
3 #include "MyKernel.hpp"
4
5 #include <pdal/Filter.hpp>
6 #include <pdal/Kernel.hpp>
7 #include <pdal/Options.hpp>
8 #include <pdal/pdal_macros.hpp>
9 #include <pdal/StageFactory.hpp>
10 #include <pdal/PointTable.hpp>
11
12 #include <memory>

```

```
13 #include <string>
14
15
16 namespace pdal {
17
18     static PluginInfo const s_info {
19         "kernels.mykernel",
20         "MyKernel",
21         "http://link/to/documentation"
22     };
23
24     CREATE_SHARED_PLUGIN(1, 0, MyKernel, Kernel, s_info);
25     std::string MyKernel::getName() const { return s_info.name; }
26
27     MyKernel::MyKernel() : Kernel()
28     {}
29
30     void MyKernel::addSwitches(ProgramArgs& args)
31     {
32         args.add("input,i", "Input filename", m_input_file).
33         →setPositional();
34         args.add("output,o", "Output filename", m_output_file).
35         →setPositional();
36     }
37
38     int MyKernel::execute()
39     {
40         PointTable table;
41         StageFactory f;
42
43         Stage& reader = makeReader(m_input_file, "readers.las");
44
45         // Options should be added in the call to makeFilter, makeReader,
46         // or makeWriter so that the system can override them with those
47         // provided on the command line when applicable.
48         Options filterOptions;
49         filterOptions.add("step", 10);
50         Stage& filter = makeFilter("filters.decimation", reader,
51         →filterOptions);
52
53         Stage& writer = makeWriter(m_output_file, filter, "writers.text
54         ");
55         writer.prepare(table);
56         writer.execute(table);
57
58         return 0;
59     }
60 }
```

```
56
57 } // namespace pdal
```

In your kernel implementation, you will use a macro defined in `pdal_macros`. This macro registers the plugin with the Kernel factory. It is only required by plugins.

```
std::string MyKernel::getName() const { return s_info.name; }
```

Note: A static plugin macro can also be used to integrate the kernel with the main code. This will not be described here. Using this as a shared plugin will be described later.

To build up a processing pipeline in this example, we need to create two objects: the `pdal::PointTable` (page 412) and the `pdal::StageFactory` (page 425). The latter is used to create the various stages that will be used within the kernel.

```
StageFactory f;
```

The `pdal::Reader` (page 420) is created from the `pdal::StageFactory` (page 425), and is specified by the stage name, in this case an LAS reader. For brevity, we provide the reader a single option, the filename of the file to be read.

```
// 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.
```

The `pdal::Filter` (page 404) is also created from the `pdal::StageFactory` (page 425). Here, we create a decimation filter that will pass every tenth point to subsequent stages. We also specify the input to this stage, which is the reader.

```
filterOptions.add("step", 10);
Stage& filter = makeFilter("filters.decimation", reader,
                           filterOptions);

Stage& writer = makeWriter(m_output_file, filter, "writers.text");
writer.prepare(table);
```

Finally, the `pdal::Writer` (page 444) is created from the `pdal::StageFactory` (page 425). This `writers.text` (page 103), takes as input the previous stage (the `filters.decimation` (page 119)) and the output filename as its sole option.

```
return 0;
}

} // namespace pdal
```

The final two steps are to prepare and execute the pipeline. This is achieved by calling `prepare` and `execute` on the final stage.

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 --help`, 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>`.

Compilation

Set up a `CMakeLists.txt` file to compile your kernel against PDAL:

```
1 cmake_minimum_required(VERSION 2.8.12)
2 project(KernelTutorial)
3
4 find_package(PDAL 1.6.0 REQUIRED CONFIG)
5
6 add_library(pdal_plugin_kernel_mykernel SHARED MyKernel.cpp)
7 target_link_libraries(pdal_plugin_kernel_mykernel PRIVATE ${PDAL_
    ↴LIBRARIES})
8 target_include_directories(pdal_plugin_kernel_mykernel PRIVATE
9                             ${PDAL_INCLUDE_DIRS})
```

12.1.8 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.

```
1 // MyReader.hpp
2
3 #pragma once
```

```

5 #include <pdal/PointView.hpp>
6 #include <pdal/Reader.hpp>
7 #include <pdal/util/IStream.hpp>
8
9 namespace pdal
10 {
11     class MyReader : public Reader
12     {
13     public:
14         MyReader() : Reader() {};
15
16         static void * create();
17         static int32_t destroy(void *);
18         std::string getName() const;
19
20     private:
21         std::unique_ptr<ILeStream> m_stream;
22         point_count_t m_index;
23         double m_scale_z;
24
25         virtual void addDimensions(PointLayoutPtr layout);
26         virtual void addArgs(ProgramArgs& args);
27         virtual void ready(PointTableRef table);
28         virtual point_count_t read(PointViewPtr view, point_count_t_
29             count);
30         virtual void done(PointTableRef table);
31     };
32 }

```

In your MyReader class, you will declare the necessary methods and variables needed to make the reader work and meet the plugin specifications.

```

1     static void * create();
2     static int32_t destroy(void *);
3     std::string getName() const;

```

These methods are required to fulfill the specs for defining a new plugin.

```

1     std::unique_ptr<ILeStream> m_stream;
2     point_count_t m_index;
3     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.

```

1     virtual void ready(PointTableRef table);
2     virtual point_count_t read(PointViewPtr view, point_count_t_
3         count);

```

```
3     virtual void done(PointTableRef table);
4 };
5 }
```

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.

```
1 // MyReader.cpp
2
3 #include "MyReader.hpp"
4 #include <pdal/pdal_macros.hpp>
5 #include <pdal/util/ProgramArgs.hpp>
6
7 namespace pdal
{
8
9     static PluginInfo const s_info = PluginInfo(
10         "readers.myreader",
11         "My Awesome Reader",
12         "http://link/to/documentation" );
13
14     CREATE_SHARED_PLUGIN(1, 0, MyReader, Reader, s_info)
15
16     std::string MyReader::getName() const { return s_info.name; }
17
18     void MyReader::addArgs(ProgramArgs& args)
19     {
20         args.add("z_scale", "Z Scaling", m_scale_z, 1.0);
21     }
22
23     void MyReader::addDimensions(PointLayoutPtr layout)
24     {
25         layout->registerDim(Dimension::Id::X);
26         layout->registerDim(Dimension::Id::Y);
27         layout->registerDim(Dimension::Id::Z);
28         layout->registerOrAssignDim("MyData",
29             Dimension::Type::Unsigned64);
30     }
31 }
```

```

30
31     void MyReader::ready(PointTableRef)
32     {
33         SpatialReference ref("EPSG:4385");
34         setSpatialReference(ref);
35     }
36
37     template <typename T>
38     T convert(const StringList& s, const std::string& name, size_t_
39     ↪fieldno)
40     {
41         T output;
42         bool bConverted = Utils::fromString(s[fieldno], output);
43         if (!bConverted)
44         {
45             std::stringstream oss;
46             oss << "Unable to convert " << name << ", " << s[fieldno] <
47             ↪< ", to double";
48             throw pdal_error(oss.str());
49         }
50
51
52         return output;
53     }
54
55     point_count_t MyReader::read(PointViewPtr view, point_count_t_
56     ↪count)
57     {
58         PointLayoutPtr layout = view->layout();
59         PointId nextId = view->size();
60         PointId idx = m_index;
61         point_count_t numRead = 0;
62
63         m_stream.reset(new ILeStream(m_filename));
64
65         size_t HEADERSIZE(1);
66         size_t skip_lines(std::max(HEADERSIZE, (size_t)m_index));
67         size_t line_no(1);
68         for (std::string line; std::getline(*m_stream->stream(), line);_
69             ↪line_no++)
69         {
70             if (line_no <= skip_lines)
71             {
72                 continue;
73             }
74
75             // MyReader format: X::Y::Z::Data

```

```

73     StringList s = Utils::split2(line, ':');
74
75     unsigned long u64(0);
76     if (s.size() != 4)
77     {
78         std::stringstream oss;
79         oss << "Unable to split proper number of fields. Expected 4,
← got "
80             << s.size();
81         throw pdal_error(oss.str());
82     }
83
84     std::string name("X");
85     view->setField(Dimension::Id::X, nextId, convert<double>(s,
←name, 0));
86
87     name = "Y";
88     view->setField(Dimension::Id::Y, nextId, convert<double>(s,
←name, 1));
89
90     name = "Z";
91     double z = convert<double>(s, name, 2) * m_scale_z;
92     view->setField(Dimension::Id::Z, nextId, z);
93
94     name = "MyData";
95     view->setField(layout->findProprietaryDim(name),
96                     nextId,
97                     convert<unsigned int>(s, name, 3));
98
99     nextId++;
100    if (m_cb)
101        m_cb(*view, nextId);
102    }
103    m_index = nextId;
104    numRead = nextId;
105
106    return numRead;
107}
108
109 void MyReader::done(PointTableRef)
110 {
111     m_stream.reset();
112 }
113
114 } //namespace pdal

```

In your reader implementation, you will use a macro defined in `pdal_macros`.

```

1 static PluginInfo const s_info = PluginInfo(
2     "readers.myreader",
3     "My Awesome Reader",
4     "http://link/to/documentation" );

```

This macro registers the plugin with the PDAL code. In this case, we are declaring this as a SHARED plugin, meaning that it will be located external to the main PDAL installation. The macro is supplied with a version number (major and minor), the class of the plugin, the parent class (in this case, to identify it as a reader), and an object with information. This information includes the name of the plugin, a description, and a link to documentation.

Creating STATIC plugins requires a few more steps which will not be covered in this tutorial.

```

1 void MyReader::addArgs(ProgramArgs& args)
2 {
3     args.add("z_scale", "Z Scaling", m_scale_z, 1.0);
4 }

```

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.

```

1 void MyReader::addDimensions(PointLayoutPtr layout)
2 {
3     layout->registerDim(Dimension::Id::X);
4     layout->registerDim(Dimension::Id::Y);
5     layout->registerDim(Dimension::Id::Z);
6     layout->registerOrAssignDim("MyData",_
7         Dimension::Type::Unsigned64);
}

```

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.

```

1 void MyReader::ready(PointTableRef)
2 {
3     SpatialReference ref("EPSG:4385");
4     setSpatialReference(ref);
5 }

```

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.

```

1 template <typename T>
2 T convert(const QStringList& s, const std::string& name, size_t_
    fieldno)

```

```

3   {
4     T output;
5     bool bConverted = Utils::fromString(s[fieldno], output);
6     if (!bConverted)
7     {
8       std::stringstream oss;
9       oss << "Unable to convert " << name << ", " << s[fieldno] <
10      << ", to double";
11      throw pdal_error(oss.str());
12    }
13
14    return output;
}

```

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 input stream.

```

1 point_count_t MyReader::read(PointViewPtr view, point_count_t
→count)

```

This method is the main processing method for the reader. It takes a pointer to a Point View 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.

```

1 size_t HEADERSIZE(1);
2 size_t skip_lines(std::max(HEADERSIZE, (size_t)m_index));
3 size_t line_no(1);

```

In preparation for reading the file, we prepare to skip some header lines. In our case, the header is only a single line.

```

1   for (std::string line; std::getline(*m_stream->stream(), line); →
2     line_no++)
3   {
4     if (line_no <= skip_lines)
5     {
6       continue;
}

```

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.

```

1      StringList s = Utils::split2(line, ':');
2
3      unsigned long u64(0);
4      if (s.size() != 4)
5      {
6          std::stringstream oss;
7          oss << "Unable to split proper number of fields. Expected 4,
8          got "
9          << s.size();
10         throw pdal_error(oss.str());
11     }

```

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.

```

1      std::string name("X");
2      view->setField(Dimension::Id::X, nextId, convert<double>(s,
3          name, 0));
4
5      name = "Y";
6      view->setField(Dimension::Id::Y, nextId, convert<double>(s,
7          name, 1));
8
9      name = "Z";
10     double z = convert<double>(s, name, 2) * m_scale_z;
11     view->setField(Dimension::Id::Z, nextId, z);
12
13     name = "MyData";
14     view->setField(layout->findProprietaryDim(name),
15                     nextId,
16                     convert<unsigned int>(s, name, 3));

```

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.

```

1      nextId++;
2      if (m_cb)
3          m_cb(*view, 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.

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.

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:

```
1 cmake_minimum_required(VERSION 2.8.12)
2 project(ReaderTutorial)
3
4 find_package(PDAL 1.6.0 REQUIRED CONFIG)
5
6 add_library(pdal_plugin_reader_myreader SHARED MyReader.cpp)
7 target_link_libraries(pdal_plugin_reader_myreader PRIVATE ${PDAL_
    ↪LIBRARIES})
8 target_include_directories(pdal_plugin_reader_myreader PRIVATE
    ${PDAL_INCLUDE_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](#) (<https://github.com/PDAL/PDAL/blob/master/examples/writing-reader/test-reader-input.txt?raw=true>) 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.

12.1.9 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.

```
1 // MyWriter.hpp
2
3 #pragma once
4
5 #include <pdal/Writer.hpp>
6
7 #include <string>
8
9 namespace pdal{
10
11     typedef std::shared_ptr<std::ostream> FileStreamPtr;
12
13     class MyWriter : public Writer
14     {
15         public:
16             MyWriter()
17             { }
18
19             static void * create();
20             static int32_t destroy(void *);
21             std::string getName() const;
22
23         private:
24             virtual void addArgs(ProgramArgs& args);
25             virtual void initialize();
26             virtual void ready(PointTableRef table);
27             virtual void write(const PointViewPtr view);
28             virtual void done(PointTableRef table);
29
30             std::string m_filename;
31             std::string m_newline;
32             std::string m_datafield;
33             int m_precision;
34
35             FileStreamPtr m_stream;
36             Dimension::Id m_dataDim;
37     };
38 }
```

```
39 } // 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.

```
1 typedef std::shared_ptr<std::ostream> FileStreamPtr;
```

FileStreamPtr is defined to make the declaration of the stream easier to manage later on.

```
static void * create();
static int32_t destroy(void *);
std::string getName() const;
```

These three methods are required to fulfill the specs for defining a new plugin.

```
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.

```
1 // MyWriter.cpp
2
3 #include "MyWriter.hpp"
4 #include <pdal/pdal_macros.hpp>
```

```

5 #include <pdal/util/FileUtils.hpp>
6 #include <pdal/util/ProgramArgs.hpp>
7
8 namespace pdal
9 {
10
11     static PluginInfo const s_info = PluginInfo(
12         "writers.mywriter",
13         "My Awesome Writer",
14         "http://path/to/documentation" );
15
16     CREATE_SHARED_PLUGIN(1, 0, MyWriter, Writer, s_info);
17
18     std::string MyWriter::getName() const { return s_info.name; }
19
20     struct FileStreamDeleter
21     {
22         template <typename T>
23         void operator()(T* ptr)
24         {
25             if (ptr)
26             {
27                 ptr->flush();
28                 FileUtils::closeFile(ptr);
29             }
30         }
31     };
32
33     void MyWriter::addArgs(ProgramArgs& args)
34     {
35         // setPositional() Makes the argument required.
36         args.add("filename", "Output filename", m_filename).
37         ↪setPositional();
38         args.add("newline", "Line terminator", m_newline, "\n");
39         args.add("datafield", "Data field", m_datafield, "UserData");
40         args.add("precision", "Precision", m_precision, 3);
41     }
42
43     void MyWriter::initialize()
44     {
45         m_stream = FileStreamPtr(FileUtils::createFile(m_filename, true),
46             FileStreamDeleter());
47         if (!m_stream)
48         {
49             std::stringstream out;
50             out << "writers.mywriter couldn't open '" << m_filename <<
51                 "' for output.";
```

```

51     throw pdal_error(out.str());
52 }
53 }
54
55 void MyWriter::ready(PointTableRef table)
56 {
57     m_stream->precision(m_precision);
58     *m_stream << std::fixed;
59
60     Dimension::Id d = table.layout()->findDim(m_datafield);
61     if (d == Dimension::Id::Unknown)
62     {
63         std::ostringstream oss;
64         oss << "Dimension not found with name '" << m_datafield << "'";
65         throw pdal_error(oss.str());
66     }
67
68     m_dataDim = d;
69
70     *m_stream << "#X:Y:Z:MyData" << m_newline;
71 }
72
73
74 void MyWriter::write(PointViewPtr view)
75 {
76     for (PointId idx = 0; idx < view->size(); ++idx)
77     {
78         double x = view->getFieldAs<double>(Dimension::Id::X, idx);
79         double y = view->getFieldAs<double>(Dimension::Id::Y, idx);
80         double z = view->getFieldAs<double>(Dimension::Id::Z, idx);
81         unsigned int myData = 0;
82
83         if (!m_datafield.empty())
84             myData = (int)(view->getFieldAs<double>(m_dataDim, idx) +
85             0.5);
86
87         *m_stream << x << ":" << y << ":" << z << ":"
88             << myData << m_newline;
89     }
90 }
91
92
93 void MyWriter::done(PointTableRef)
94 {
95     m_stream.reset();
96 }
```

```
97
98 }
```

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 = PluginInfo(
    "writers.mywriter",
    "My Awesome Writer",
    "http://path/to/documentation" );

CREATE_SHARED_PLUGIN(1, 0, MyWriter, Writer, 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 plugin, which means it will be external to the main PDAL installation. When using the macro, we specify the version (major and minor), the class of the plugin, the class of the parent (Writer, in this case), and the struct we defined earlier.

Creating STATIC plugins, which would be part of the main PDAL installation, is also possible, but requires some extra steps and will not be covered here.

```
1  struct FileStreamDeleter
2  {
3      template <typename T>
4      void operator()(T* ptr)
5      {
6          if (ptr)
7          {
8              ptr->flush();
9              FileUtils::closeFile(ptr);
10         }
11     }
12 };
```

This struct is used for helping with the `FileStreamPtr` for cleanup.

```
1  void MyWriter::addArgs(ProgramArgs& args)
2  {
3      // setPositional() Makes the argument required.
4      args.add("filename", "Output filename", m_filename).
5      ↪setPositional();
6      args.add("newline", "Line terminator", m_newline, "\n");
7      args.add("datafield", "Data field", m_datafield, "UserData");
8      args.add("precision", "Precision", m_precision, 3);
9  }
```

This method defines the arguments the writer provides and binds them to private variables.

```

1   {
2     std::stringstream out;
3     out << "writers.mywriter couldn't open '" << m_filename <<
4       "' for output.";
5     throw pdal_error(out.str());
6   }
7 }

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;

```

This method initializes our file stream in preparation for writing.

```

1 void MyWriter::ready(PointTableRef table)
2 {
3   m_stream->precision(m_precision);
4   *m_stream << std::fixed;
5
6   Dimension::Id d = table.layout()->findDim(m_datafield);
7   if (d == Dimension::Id::Unknown)
8   {
9     std::ostringstream oss;
10    oss << "Dimension not found with name '" << m_datafield << "'";
11    throw pdal_error(oss.str());
12  }
13
14  m_dataDim = d;
15
16  *m_stream << "#X:Y:Z:MyData" << m_newline;

```

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.

```

1 void MyWriter::write(PointViewPtr view)
2 {
3   for (PointId idx = 0; idx < view->size(); ++idx)
4   {
5     double x = view->getFieldAs<double>(Dimension::Id::X, idx);

```

```

6     double y = view->getFieldAs<double>(Dimension::Id::Y, idx);
7     double z = view->getFieldAs<double>(Dimension::Id::Z, idx);
8     unsigned int myData = 0;
9
10    if (!m_datafield.empty()) {
11        myData = (int)(view->getFieldAs<double>(m_dataDim, idx) +
12            0.5);
13    }
14
15    *m_stream << x << ":" << y << ":" << z << ":" <<
16    myData << m_newline;
17}

```

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.

```

1 void MyWriter::done(PointTableRef)
2 {
3     m_stream.reset();
4 }

```

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:

```

1 cmake_minimum_required(VERSION 2.8.12)
2 project(WriterTutorial)
3
4 find_package(PDAL 1.6.0 REQUIRED CONFIG)
5
6 add_library(pdal_plugin_writer_mywriter SHARED MyWriter.cpp)

```

```
7 target_link_libraries(pdal_plugin_writer_mywriter PRIVATE ${PDAL_
8   ↪LIBRARIES})
9 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 convert 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.

12.1.10 CMake

Author Bradley Chambers

Contact brad.chambers@gmail.com

Date 01/21/2015

This tutorial will explain how to use PDAL in your own projects using CMake. A more complete, working example can be found [here](#) (page 355).

Note: We assume you have either *built or installed* (page 334) PDAL.

Basic CMake configuration

Begin by creating a file named `CMakeLists.txt` that contains:

```
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
```

12.2 API

PDAL is a C++ library, and its primary API is in that language. There is also a [Python](#) (page 189) 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 23) and users looking for documentation on how to contribute to PDAL should look [here](#) (page 323).

12.2.1 C++ API

`pdal::BOX2D`

class pdal::BOX2D

BOX2D (page 387) represents a two-dimensional box with double-precision bounds.

Subclassed by *pdal::BOX3D* (page 390)

Public Functions

BOX2D ()

Construct an “empty” bounds box.

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.

Return Whether the bounds box is empty.

bool valid () const

Determine whether a bounds box has had any bounds set.

Return Whether the bounds box is valid.

void clear ()

Clear the bounds box to an empty state.

void grow (double *x*, double *y*)

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.

`bool contains (double x, double y) const`

Determine if a bounds box contains a point.

Return Whether both dimensions are equal to or less than the maximum box values and equal to or more than the minimum box values.

Parameters

- x: X dimension value.
- y: Y dimension value.

`bool equal (const BOX2D (page 387) &other) const`

Determine if the bounds of this box are the same as that of another box.

Empty bounds boxes are always equal.

Return `true` if the provided box has equal limits to this box, `false` otherwise.

Parameters

- other: Bounds box to check for equality.

`bool operator==(BOX2D (page 387) const &other) const`

Determine if the bounds of this box are the same as that of another box.

Empty bounds boxes are always equal.

Return `true` if the provided box has equal limits to this box, `false` otherwise.

Parameters

- other: Bounds box to check for equality.

`bool operator!=(BOX2D (page 387) const &other) const`

Determine if the bounds of this box are different from that of another box.

Empty bounds boxes are never unequal.

Return `true` if the provided box has limits different from this box, `false` otherwise.

Parameters

- other: Bounds box to check for inequality.

void **grow** (**const BOX2D** (page 387) &*other*)
Expand this box to contain another box.

Parameters

- *other*: Box that this box should contain.

void **clip** (**const BOX2D** (page 387) &*other*)
Clip this bounds box by another so it will be contained by the other box.

Parameters

- *other*: Clipping box for this box.

bool **contains** (**const BOX2D** (page 387) &*other*) **const**
Determine if another bounds box is contained in this bounds box.

Equal limits are considered to be contained.

Return `true` if the provided box is contained in this box, `false` otherwise.

Parameters

- *other*: Bounds box to check for containment.

bool **overlaps** (**const BOX2D** (page 387) &*other*)
Determine if another box overlaps this box.

Return Whether the provided box overlaps this box.

Parameters

- *other*: Box to test for overlap.

std::string **toBox** (uint32_t *precision* = 8) **const**
Convert this box to a string suitable for use in SQLite.

Return String format of this box.

Parameters

- *precision*: Precision for output [default: 8]

std::string **toWKT** (uint32_t *precision* = 8) **const**
Convert this box to a well-known text string.

Return String format of this box.

Parameters

- **precision:** Precision for output [default: 8]

```
std::string toGeoJSON (uint32_t precision = 8) const  
Convert this box to a GeoJSON text string.
```

Return String format of this box.

Parameters

- **precision:** Precision for output [default: 8]

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

const BOX2D (page 387) &**getDefaultSpatialExtent** ()

Return a statically-allocated Bounds extent that represents infinity.

Return A bounds box with infinite bounds,

class pdal::BOX3D

BOX3D (page 390) represents a three-dimensional box with double-precision bounds.

Inherits from *pdal::BOX2D* (page 387)

Public Functions

BOX3D ()

Clear the bounds box to an empty state.

BOX3D (const BOX3D (page 390) &*box*)

BOX3D (const BOX2D (page 387) &*box*)

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.

bool **empty () const**

Determine whether a bounds box has not had any bounds set (is in a state as if default-constructed).

Return Whether the bounds box is empty.

bool **valid () const**

Determine whether a bounds box has had any bounds set.

Return if the bounds box is not empty

void **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.

bool **contains** (double *x*, double *y*, double *z*) **const**

Determine if a bounds box contains a point.

Return Whether both dimensions are equal to or less than the maximum box values and equal to or more than the minimum box values.

Parameters

- `x`: X dimension value.
- `y`: Y dimension value.
- `z`: Z dimension value.

`bool contains (const BOX3D (page 390) &other) const`

Determine if another bounds box is contained in this bounds box.

Equal limits are considered to be contained.

Return `true` if the provided box is contained in this box, `false` otherwise.

Parameters

- `other`: Bounds box to check for containment.

`bool equal (const BOX3D (page 390) &other) const`

Determine if the bounds of this box are the same as that of another box.

Empty bounds boxes are always equal.

Return `true` if the provided box has equal limits to this box, `false` otherwise.

Parameters

- `other`: Bounds box to check for equality.

`bool operator==(BOX3D (page 390) const &rhs) const`

Determine if the bounds of this box are the same as that of another box.

Empty bounds boxes are always equal.

Return `true` if the provided box has equal limits to this box, `false` otherwise.

Parameters

- `other`: Bounds box to check for equality.

`bool operator!=(BOX3D (page 390) const &rhs) const`

Determine if the bounds of this box are different from that of another box.

Empty bounds boxes are never unequal.

Return `true` if the provided box has limits different from this box, `false` otherwise.

Parameters

- `other`: Bounds box to check for inequality.

`void grow (const BOX3D (page 390) &other)`

Expand this box to contain another box.

Parameters

- `other`: Box that this box should contain.

`void clip (const BOX3D (page 390) &other)`

Clip this bounds box by another so it will be contained by the other box.

Parameters

- `other`: Clipping box for this box.

`bool overlaps (const BOX3D (page 390) &other)`

Determine if another box overlaps this box.

Return Whether the provided box overlaps this box.

Parameters

- `other`: Box to test for overlap.

`BOX2D (page 387) to2d() const`

Convert this box to 2-dimensional bounding box.

Return Bounding box with Z dimension stripped.

`std::string toBox (uint32_t precision = 8) const`

Convert this box to a string suitable for use in SQLite.

Return String format of this box.

Parameters

- `precision`: Precision for output [default: 8]

`std::string toWKT (uint32_t precision = 8) const`

Convert this box to a well-known text string.

Return String format of this box.

Parameters

- precision: Precision for output [default: 8]

Public Members

double minz
Minimum Z value.

double maxz
Maximum Z value.

Public Static Functions

const *BOX3D* (page 390) &getDefaultSpatialExtent ()
Return a statically-allocated Bounds extent that represents infinity.

Return A bounds box with infinite bounds,

pdal::Charbuf

class pdal::Charbuf
Allow a data buffer to be used at a std::streambuf.
Inherits from streambuf

Public Functions

PDAL_DLL Charbuf ()
Construct an empty *Charbuf* (page 394).

PDAL_DLL Charbuf (std::vector<char> &v, pos_type bufOffset = 0)
Construct a *Charbuf* (page 394) that wraps a byte vector.

Parameters

- v: Byte vector to back streambuf.
- bufOffset: Offset in vector (ignore bytes before offset).

PDAL_DLL Charbuf (char *buf, size_t count, pos_type bufOffset = 0)
Construct a *Charbuf* (page 394) that wraps a byte buffer.

Parameters

- `buf`: Buffer to back streambuf.
- `count`: Size of buffer.
- `bufOffset`: Offset in vector (ignore bytes before offset).

```
void initialize (char *buf, size_t count, pos_type bufOffset = 0)  
    Set a buffer to back a Charbuf (page 394).
```

Parameters

- `buf`: Buffer to back streambuf.
- `count`: Size of buffer.
- `bufOffset`: Offset in vector (ignore bytes before offset).

`pdal::Dimension`

```
namespace pdal::Dimension
```

TypeDefs

```
typedef std::vector<Detail> DetailList
```

Enums

```
enum BaseType
```

Values:

None = 0x000

Signed = 0x100

Unsigned = 0x200

Floating = 0x400

```
enum Type
```

Values:

None = 0

Unsigned8 = unsigned(BaseType::Unsigned) | 1

Signed8 = unsigned(BaseType::Signed) | 1

Unsigned16 = unsigned(BaseType::Unsigned) | 2

```
Signed16 = unsigned(BaseType::Signed) | 2
Unsigned32 = unsigned(BaseType::Unsigned) | 4
Signed32 = unsigned(BaseType::Signed) | 4
Unsigned64 = unsigned(BaseType::Unsigned) | 8
Signed64 = unsigned(BaseType::Signed) | 8
Float = unsigned(BaseType::Floating) | 4
Double = unsigned(BaseType::Floating) | 8
```

Functions

BaseType (page 395) **fromName** (std::string *name*)

std::string **toName** (*BaseType* (page 395) *b*)

std::size_t **size** (*Type* (page 395) *t*)

BaseType (page 395) **base** (*Type* (page 395) *t*)

std::string **interpretationName** (*Type* (page 395) *dimtype*)

Get a string representation of a datatype.

Return String representation of dimension type.

Parameters

- *dimtype*: *Dimension* (page 395) type.

Type (page 395) **type** (std::string *s*)

Get the type corresponding to a type name.

Return Corresponding type enumeration value.

Parameters

- *s*: Name of type.

std::size_t **extractName** (**const** std::string &*s*, std::string::size_type *p*)

Extract a dimension name of a string.

Dimension (page 395) names start with an alpha and continue with numbers or underscores.

Return Number of characters in the extracted name.

Parameters

- *s*: String from which to extract dimension name.
- *p*: Position at which to start extracting.

```
std::istream &operator>> (std::istream &in, Dimension (page 395)::Type  
                           (page 395) &type)
```

```
std::ostream &operator<< (std::ostream &out, const Dimension  
                           (page 395)::Type (page 395) &type)
```

Variables

```
const int COUNT = std::numeric_limits<uint16_t>::max()
```

```
const int PROPRIETARY = 0xF000
```

pdal::Extractor

class pdal::Extractor

Buffer wrapper for input of binary data from a buffer.

Subclassed by pdal::BeExtractor, pdal::LeExtractor, pdal::SwitchableExtractor

Public Functions

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.

operator bool()

Determine if the buffer is good.

Return Whether the buffer is good.

void seek (std::size_t pos)

Seek to a position in the buffer.

Parameters

- *pos*: Position to seek in buffer.

void skip (std::size_t cnt)
Advance buffer position.

Parameters

- `cnt`: Number of bytes to skip in buffer.

size_t position () const
Return the get position of buffer.

Return Get position.

bool good () const
Determine whether the extractor is good (the get pointer is in the buffer).

Return Whether the get pointer is valid.

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.

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.

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.

```
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.

```
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 397) &operator>> (uint8_t &v) = 0
```

```
virtual Extractor (page 397) &operator>> (int8_t &v) = 0
```

```
virtual Extractor (page 397) &operator>> (uint16_t &v) = 0
```

```
virtual Extractor (page 397) &operator>> (int16_t &v) = 0
```

```
virtual Extractor (page 397) &operator>> (uint32_t &v) = 0
```

```
virtual Extractor (page 397) &operator>> (int32_t &v) = 0
```

```
virtual Extractor (page 397) &operator>> (uint64_t &v) = 0
```

```
virtual Extractor (page 397) &operator>> (int64_t &v) = 0
```

```
virtual Extractor (page 397) &operator>> (float &v) = 0
```

```
virtual Extractor (page 397) &operator>> (double &v) = 0
```

```
pdal::FileUtils
```

```
namespace pdal::FileUtils
```

Functions

```
PDAL_DLL std::istream * pdal::FileUtils::openFile(std::string const &
    Open an existing file for reading.
```

Return Pointer to opened stream.

Parameters

- filename: Filename.
- asBinary: Read as binary file (don't convert /r/n to /n)

PDAL_DLL std::ostream * pdal::FileUtils::createFile(const std::string const &
Create a file and open for writing.

Return Point to opened stream.

Parameters

- filename: Filename.
- asBinary: Write as binary file (don't convert /n to /r/n)

PDAL_DLL bool pdal::FileUtils::directoryExists(const std::string & dirname)
Determine if a directory exists.

Return Whether a directory exists.

Parameters

- dirname: Name of directory.

PDAL_DLL bool pdal::FileUtils::createDirectory(const std::string & dirname)
Create a directory.

Return Whether the directory was created.

Parameters

- dirname: Directory name.

PDAL_DLL void pdal::FileUtils::deleteDirectory(const std::string & dirname)
Delete a directory and its contents.

Parameters

- dirname: Directory name.

PDAL_DLL std::vector< std::string > pdal::FileUtils::directoryList(const std::string &
List the contents of a directory.

Return List of entries in the directory.

Parameters

- dirname: Name of directory to list.

```
PDAL_DLL void pdal::FileUtils::closeFile(std::ostream * ofs)
    Close a file created with createFile.
```

Parameters

- `ofs`: Pointer to stream to close.

```
PDAL_DLL void pdal::FileUtils::closeFile(std::istream * ifs)
    Close a file created with openFile.
```

Parameters

- `ifs`: Pointer to stream to close.

```
PDAL_DLL bool pdal::FileUtils::deleteFile(const std::string & filename)
    Delete a file.
```

Return `true` if successful, `false` otherwise

Parameters

- `filename`: Name of file to delete.

```
PDAL_DLL void pdal::FileUtils::renameFile(const std::string & dest, const std::string & src)
    Rename a file.
```

Parameters

- `dest`: Desired filename.
- `src`: Source filename.

```
PDAL_DLL bool pdal::FileUtils::fileExists(const std::string & filename)
    Determine if a file exists.
```

Return Whether the file exists.

Parameters

- `filename..`:

```
PDAL_DLL uintmax_t pdal::FileUtils::fileSize(const std::string & filename)
    Get the size of a file.
```

Return Size of file.

Parameters

- `filename`: Filename.

```
PDAL_DLL std::string pdal::FileUtils::readFileToString(const std::string & filename)
    Read a file into a string.
```

Return File contents as a string

Parameters

- `filename`: Filename.

PDAL_DLL std::string pdal::FileUtils::getcwd()

Get the current working directory with trailing separator.

Return The current working directory.

PDAL_DLL std::string pdal::FileUtils::toAbsolutePath(const std::string &

If the filename is an absolute path, just return it otherwise, make it absolute (relative to current working dir) and return it.

Return Absolute version of provided filename.

Parameters

- `filename`: Name of file to convert to absolute path.

PDAL_DLL std::string pdal::FileUtils::toAbsolutePath(const std::string &

If the filename is an absolute path, just return it otherwise, make it absolute (relative to base dir) and return that.

Return Absolute version of provided filename relative to base.

Parameters

- `filename`: Name of file to convert to absolute path.
- `base`: Base name to use.

PDAL_DLL std::string pdal::FileUtils::getFilename(const std::string & p)

Return the file component of the given path, e.g.

“d:/foo/bar/a.c” -> “a.c”

Return File part of path.

Parameters

- `path`: Path from which to extract file component.

PDAL_DLL std::string pdal::FileUtils::getDirectory(const std::string & p)

Return the directory component of the given path, e.g.

“d:/foo/bar/a.c” -> “d:/foo/bar/”

Return Directory part of path.

Parameters

- `path`: Path from which to extract directory component.

PDAL_DLL std::string pdal::FileUtils::stem(const std::string & path)

Return the filename stripped of the extension.

. and .. are returned unchanged.

Return Stem of filename.

Parameters

- path: File path from which to extract file stem.

PDAL_DLL bool pdal::FileUtils::isDirectory(const std::string & path)

Determine if path is a directory.

Return Whether the path represents a directory.

Parameters

- path: Directory to check.

PDAL_DLL bool pdal::FileUtils::isAbsolutePath(const std::string & path)

Determine if the path is an absolute path.

Return Whether the path is absolute.

Parameters

- path: Path to test.

PDAL_DLL void pdal::FileUtils::fileTimes(const std::string & filename,

Get the file creation and modification times.

Parameters

- filename: Filename.
- createTime: Pointer to creation time structure.
- modTime: Pointer to modification time structure.

PDAL_DLL std::string pdal::FileUtils::extension(const std::string & path)

Return the extension of the filename, including the separator (.).

Return Extension of filename.

Parameters

- path: File path from which to extract extension.

PDAL_DLL std::vector< std::string > pdal::FileUtils::glob(std::string path)

Expand a filespec to a list of files.

Return List of files that correspond to provided file specification.

Parameters

- `fileSpec`: File specification to expand.

`pdal::Filter`

`class pdal::Filter`

Inherits from [*pdal::Stage*](#) (page 420)

Subclassed by `pdal::ApproximateCoplanarFilter`, `pdal::AssignFilter`, `pdal::ChipperFilter`, `pdal::ClusterFilter`, `pdal::ColorinterpFilter`, `pdal::ColorizationFilter`, `pdal::ComputeRangeFilter`, `pdal::CpdFilter`, `pdal::CropFilter`, `pdal::DecimationFilter`, `pdal::DividerFilter`, `pdal::EigenvaluesFilter`, `pdal::ELMFilter`, `pdal::EstimateRankFilter`, `pdal::FerryFilter`, `pdal::GreedyProjection`, `pdal::GreedyProjectionFilter`, `pdal::GridProjectionFilter`, `pdal::GroupByFilter`, `pdal::HAGFilter`, `pdal::HeadFilter`, `pdal::HexBin`, `pdal::IcpFilter`, `pdal::IQRFilter`, `pdal::KDistanceFilter`, `pdal::LocateFilter`, `pdal::LOFFilter`, `pdal::MADFilter`, `pdal::MatlabFilter`, `pdal::MergeFilter`, `pdal::MongusFilter`, `pdal::MortonOrderFilter`, `pdal::MovingLeastSquaresFilter`, `pdal::NormalFilter`, `pdal::OutlierFilter`, `pdal::OverlayFilter`, `pdal::PCLBlock`, `pdal::PMFFilter`, `pdal::PoissonFilter`, `pdal::PoissonFilter`, `pdal::PythonFilter`, `pdal::RadialDensityFilter`, `pdal::RandomizeFilter`, `pdal::RangeFilter`, `pdal::ReprojectionFilter`, `pdal::SampleFilter`, `pdal::SMRFilter`, `pdal::SortFilter`, `pdal::SplitterFilter`, `pdal::StatsFilter`, `pdal::StreamCallbackFilter`, `pdal::TailFilter`, `pdal::TransformationFilter`, `pdal::VoxelCenterNearestNeighborFilter`, `pdal::VoxelCentroidNearestNeighborFilter`, `pdal::VoxelGridFilter`, `SplitFilter`

Public Functions

`Filter()`

`pdal::IStream`

`class pdal::IStream`

Stream wrapper for input of binary data.

Subclassed by `pdal::IBeStream`, `pdal::ILeStream`, `pdal::ISwitchableStream`

Public Functions

`PDAL_DLL IStream()`

Default constructor.

PDAL_DLL *IStream*(const std::string &filename)

Construct an *IStream* (page 404) from a filename.

Parameters

- filename: File from which to read.

PDAL_DLL *IStream*(std::istream *stream)

Construct an *IStream* (page 404) from an input stream pointer.

Parameters

- stream: Stream from which to read.

PDAL_DLL ~*IStream*()

PDAL_DLL int pdal::IStream::open(const std::string & filename)

Open a file to extract.

Return -1 if a stream is already assigned, 0 otherwise.

Parameters

- filename: Filename.

PDAL_DLL void pdal::IStream::close()

Close the underlying stream.

PDAL_DLL operator bool()

Return the state of the stream.

Return The state of the underlying stream.

PDAL_DLL void pdal::IStream::seek(std::streampos pos)

Seek to a position in the underlying stream.

Parameters

- pos: Position to seek to,

PDAL_DLL void pdal::IStream::seek(std::streampos off, std::ios_base::

Seek to an offset from a specified position.

Parameters

- off: Offset.
- way: Absolute position for offset (beg, end or cur)

PDAL_DLL void pdal::IStream::skip(std::streamoff offset)

Skip relative to the current position.

Parameters

- offset: Offset from the current position.

PDAL_DLL std::streampos pdal::IStream::position() const
Determine the position of the get pointer.

Return Current get position.

PDAL_DLL bool pdal::IStream::good() const
Determine if the underlying stream is good.

Return Whether the underlying stream is good.

PDAL_DLL std::istream* pdal::IStream::stream()
Fetch a pointer to the underlying stream.

Return Pointer to the underlying stream.

PDAL_DLL void pdal::IStream::pushStream(std::istream * strm)
Temporarily push a stream to read from.

Parameters

- strm: New stream to read from.

PDAL_DLL std::istream* pdal::IStream::popStream()
Pop the current stream and return it.

The last stream on the stack cannot be popped.

Return Pointer to the popped stream.

PDAL_DLL void pdal::IStream::get(std::string & s, size_t size)
Fetch data from the stream into a string.

Parameters

- s: String to fill.
- size: Number of bytes to extract.

PDAL_DLL void pdal::IStream::get(std::vector< char > & buf)
Fetch data from the stream into a vector of char.

Parameters

- buf: Buffer to fill.

PDAL_DLL void pdal::IStream::get(std::vector< unsigned char > & buf)
Fetch data from the stream into a vector of unsigned char.

Parameters

- buf: Buffer to fill.

PDAL_DLL void pdal::IStream::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.

PDAL_DLL void pdal::IStream::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 407) is a logging object that is provided by *pdal::Stage* (page 420) to facilitate logging operations.

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

LogLevel **getLevel()**

Return the logging level of the *pdal::Log* (page 407) instance

void **setLevel(LogLevel v)**

Sets the logging level of the *pdal::Log* (page 407) instance.

Parameters

- v: logging level to use for *get()* (page 408) comparison operations

`void setLeader (const std::string &leader)`
Set the leader string (deprecated).

Parameters

- `leader`: Leader string.

`void pushLeader (const std::string &leader)`
Push the leader string onto the stack.

Parameters

- `leader`: Leader string

`std::string leader () const`
Get the leader string.

Return The current leader string.

`void popLeader ()`
Pop the current leader string.

`std::string getLogLevel (LogLevel v) const`

Return A string representing the LogLevel

Log stream operations

`std::ostream *getLogStream ()`

Return the stream object that is currently being used to 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 408) is less than the logging level of the `pdal::Log` (page 407) instance

`void floatPrecision (int level)`
Sets the floating point precision.

```
void clearFloat()
```

Clears the floating point precision settings of the streams.

Public Functions

```
Log(std::string const &leaderString, std::string const &outputName)
```

Constructs a *pdal::Log* (page 407) instance.

Parameters

- *leaderString*: A string to presage all log entries with
- *outputName*: A filename or one of ‘stdout’, ‘stdlog’, or ‘stderr’ to use for outputting log information.

```
Log(std::string const &leaderString, std::ostream *v)
```

Constructs a *pdal::Log* (page 407) instance.

Parameters

- *leaderString*: A string to presage all log entries with
- *v*: An existing std::ostream to use for logging (instead of the the instance creating its own)

```
pdal::Metadata
```

```
class pdal::Metadata
```

Public Functions

```
Metadata()
```

```
Metadata(const std::string &name)
```

```
MetadataNode (page 409) getNode() const
```

Public Static Functions

```
std::string inferType(const std::string &val)
```

```
class pdal::MetadataNode
```

Public Functions

MetadataNode ()

MetadataNode (const std::string &name)

MetadataNode (page 409) **add (const std::string &name)**

MetadataNode (page 409) **addList (const std::string &name)**

MetadataNode (page 409) **clone (const std::string &name) const**

MetadataNode (page 409) **add (MetadataNode (page 409) node)**

MetadataNode (page 409) **addList (MetadataNode (page 409) node)**

MetadataNode (page 409) **addEncoded (const std::string &name, const unsigned char *buf, size_t size, const std::string &descrip = std::string())**

MetadataNode (page 409) **addListEncoded (const std::string &name, const unsigned char *buf, size_t size, const std::string &descrip = std::string())**

MetadataNode (page 409) **addWithType (const std::string &name, const std::string &value, const std::string &type, const std::string &descrip)**

template <typename T>

MetadataNode (page 409) **add (const std::string &name, const T &value, const std::string &descrip = std::string())**

template <typename T>

MetadataNode (page 409) **addList (const std::string &name, const T &value, const std::string &descrip = std::string())**

template <typename T>

MetadataNode (page 409) **addOrUpdate (const std::string &lname, const T &value)**

template <typename T>

MetadataNode (page 409) **addOrUpdate (const std::string &lname, const T &value, const std::string &descrip)**

std::string type () const

MetadataType kind () const

std::string name () const

template <typename T>

```
T value() const
std::string value() const
std::string jsonValue() const
std::string description() const
MetadataNodeList children() const
MetadataNodeList children(const std::string &name) const
bool hasChildren() const
std::vector<std::string> childNames() const
bool operator!()
bool valid() const
bool empty() const
template <typename PREDICATE>
MetadataNode (page 409) find(PREDICATE p) const
template <typename PREDICATE>
MetadataNodeList findChildren(PREDICATE p)
template <typename PREDICATE>
MetadataNode (page 409) findChild(PREDICATE p) const
MetadataNode (page 409) findChild(const char *s) const
MetadataNode (page 409) findChild(std::string s) const
```

pdal::Options

class pdal::Options

Public Functions

```
Options()
Options(const Option &opt)
void add(const Option &option)
void add(const Options (page 411) &options)
void addConditional(const Option &option)
```

```
void addConditional (const Options (page 411) &option)
void remove (const Option &option)
void replace (const Option &option)
void toMetadata (MetadataNode (page 409) &parent) const
template <typename T>
void add (const std::string &name, T value)
void add (const std::string &name, const std::string &value)
void add (const std::string &name, const bool &value)
template <typename T>
void replace (const std::string &name, T value)
void replace (const std::string &name, const std::string &value)
void replace (const std::string &name, const bool &value)
StringList getValues (const std::string &name) const
StringList getKeys () const
std::vector<Option> getOptions (std::string const &name = "") const
StringList toCommandLine () const
    Convert options to a string list appropriate for parsing with ProgramArgs
    (page 416).
```

Return List of options as argument strings.

pdal::PointTable

```
class pdal::PointTable
Inherits from pdal::SimplePointTable
```

Public Functions

```
PointTable()
~PointTable()
virtual bool supportsView() const
```

`pdal::PointView`

`class pdal::PointView`
Inherits from `pdal::PointContainer`

Public Functions

`PointView (const PointView (page 413)&)`

`PointView (page 413) &operator= (const PointView (page 413)&)`

`PointView (PointTableRef pointTable)`

`PointView (PointTableRef pointTable, const SpatialReference &srs)`

`~PointView ()`

`PointViewIter begin ()`

`PointViewIter end ()`

`int id () const`

`point_count_t size () const`

`bool empty () const`

`void appendPoint (const PointView (page 413) &buffer, PointId id)`

`void append (const PointView (page 413) &buf)`

`PointViewPtr makeNew () const`

Return a new point view with the same point table as this point buffer.

`PointRef point (PointId id)`

`template <class T>`

`T getFieldAs (Dimension (page 395)::Id dim, PointId pointIndex) const`

`void getField (char *pos, Dimension (page 395)::Id d, Dimension (page 395)::Type (page 395) type, PointId id) const`

`template <typename T>`

`void setField (Dimension (page 395)::Id dim, PointId idx, T val)`

`void setField (Dimension (page 395)::Id dim, Dimension (page 395)::Type (page 395) type, PointId idx, const void *val)`

`template <typename T>`

`bool compare (Dimension (page 395)::Id dim, PointId id1, PointId id2)`

```
bool compare (Dimension (page 395)::Id dim, PointId id1, PointId id2)  
void getRawField (Dimension (page 395)::Id dim, PointId idx, void *buf)  
    const  
void calculateBounds (BOX2D (page 387) &box) const
```

Return a cumulated bounds of all points in the *PointView* (page 413).

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.

```
void calculateBounds (BOX3D (page 390) &box) const  
void dump (std::ostream &ostr) const  
bool hasDim (Dimension (page 395)::Id id) const  
std::string dimName (Dimension (page 395)::Id id) const  
Dimension (page 395)::IdList dims () const  
std::size_t pointSize () const  
std::size_t dimSize (Dimension (page 395)::Id id) const  
Dimension (page 395)::Type (page 395) dimType (Dimension (page 395)::Id id)  
    const  
DimTypeList dimTypes () const  
PointLayoutPtr layout () const  
void setSpatialReference (const SpatialReference &spatialRef)  
SpatialReference spatialReference () const  
void getPackedPoint (const DimTypeList &dims, PointId idx, char *buf)  
    const  
Fill a buffer with point data specified by the dimension list.
```

Parameters

- *dims*: List of dimensions/types to retrieve.
- *idx*: Index of point to get.
- *buf*: Pointer to buffer to fill.

```
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

- dims: Dimension/types of data in packed order
- idx: Index of point to write.
- buf: Packed data buffer.

```
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.

```
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.

```
void clearTemps ()
```

MetadataNode (page 409) **toMetadata** () **const**

```
TriangularMesh *createMesh (const std::string &name)
```

Creates a mesh with the specified name.

Return Pointer to the new mesh. Null is returned if the mesh already exists.

Parameters

- name: Name of the mesh.

```
TriangularMesh *mesh (const std::string &name = "")
```

Get a pointer to a mesh.

Return New mesh. Null is returned if the mesh already exists.

Parameters

- name: Name of the mesh.

```
KD3Index &build3dIndex ()
```

```
KD2Index &build2dIndex ()
```

Public Static Functions

`void calculateBounds (const PointViewSet &set, BOX2D (page 387) &box)`

`void calculateBounds (const PointViewSet &set, BOX3D (page 390) &box)`

Friends

`friend pdal::PointView::plang::Invocation`

`pdal::ProgramArgs`

`class pdal::ProgramArgs`

Parses command lines, provides validation and stores found values in bound variables.

Add arguments with `add` (page 416). When all arguments have been added, use `parse` (page 418) to validate command line and assign values to variables bound with `add` (page 416).

Public Functions

`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.

Return Reference to the new 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.
- `def`: Default value of argument.

`Arg &add (const std::string &name, const std::string &description, std::vector<std::string> &var)`

Add a list-based (vector) string argument.

Return Reference to the new 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.

bool `set (const std::string &name) const`

Return whether the argument (as specified by its longname) had its value set during parsing.

template <typename T>

Arg &`add (const std::string &name, const std::string &description,`
`std::vector<T> &var)`

Add a list-based (vector) argument.

Return Reference to the new 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.

template <typename T>

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.

Return Reference to the new 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.

template <typename T>

Arg &`add (const std::string &name, const std::string description, T &var, T`
`def)`

Add an argument to the list of arguments with a default.

Return Reference to the new 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.
- **def**: Default value of argument.

template <typename T>

Arg &add (const std::string &name, const std::string description, T &var)

Add an argument to the list of arguments.

Return Reference to the new 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.

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.

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.

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.

void reset ()
Reset the state of all arguments and bound variables as if no parsing had occurred.

std::string commandLine () const
Return a string suitable for use in a “usage” line for display to users as help.

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.

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.

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 420) 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 pdal::Reader`

Inherits from `pdal::Stage` (page 420)

Subclassed by `pdal::BpfReader`, `pdal::BufferReader`, `pdal::DbReader`, `pdal::FauxReader`, `pdal::GDALReader`, `pdal::GeoWaveReader`, `pdal::GreyhoundReader`, `pdal::IcebridgeReader`, `pdal::Ilvis2Reader`, `pdal::LasReader`, `pdal::MatlabReader`, `pdal::MbReader`, `pdal::MrsidReader`, `pdal::OptechReader`, `pdal::OSGReader`, `pdal::PcdReader`, `pdal::PlyReader`, `pdal::PtsReader`, `pdal::QfitReader`, `pdal::RxpReader`, `pdal::SbetReader`, `pdal::TerrasolidReader`, `pdal::TextReader`, `pdal::TIndexReader`

`pdal::Stage`

`pdal::Stage` (page 420) is the base class of `pdal::Filter` (page 404), `pdal::Reader` (page 420), 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 420) into a pipeline. The pipeline is run with by calling in sequence `prepare()` (page 421) and `execute()` (page 421) 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 404), `pdal::Reader` (page 420), `pdal::Writer` (page 444)

Public Functions

`Stage()`

`virtual ~Stage()`

`void setInput (Stage (page 420) &input)`

Add a stage to the input list of this stage.

Parameters

- `input`: *Stage* (page 420) to use as input.

`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 421) can be called. Prepare recurses through all input stages.

Parameters

- `table`: *PointTable* (page 412) 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 421) function.

`void execute (StreamPointTable &table)`

Execute a prepared pipeline (linked set of stages) in streaming mode.

This performs the action associated with the stage by executing the `processOne` function of each stage in depth first order. Points are processed up to the capacity of the provided `StreamPointTable`. Not all stages support streaming mode and an exception will be thrown when attempting to `execute` (page 421) an unsupported stage.

Streaming points can reduce memory consumption, but may limit access to algorithms that need to operate on full point sets.

Parameters

- `table`: Streaming point table used for stage pipeline. This must be the same table used in the [prepare](#) (page 421) function.

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 413) 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.

Parameters

- `srs`: Spatial reference to set.

`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 413) 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.

Return The stage's spatial reference.

void `setOptions` (`Options` (page 411) `options`)

Set a stage's options.

Set the options on a stage, clearing all previously set options.

Parameters

- `options`: [Options](#) (page 411) to set.

void `addConditionalOptions` (`const Options` (page 411) `&opts`)

Add options if an option with the same name doesn't already exist on the stage.

Parameters

- `opts`: [Options](#) (page 411) to add.

void `addAllArgs` (*ProgramArgs* (page 416) &*args*)
Add a stage's options to a *ProgramArgs* (page 416) set.

Parameters

- *args*: *ProgramArgs* (page 416) to add to.

void `addOptions` (const Options** (page 411) &*opts*)**
Add options to the existing option set.

Parameters

- *opts*: *Options* (page 411) to add.

void `removeOptions` (const Options** (page 411) &*opts*)**
Remove options from a stage's option set.

Parameters

- *opts*: *Options* (page 411) to remove.

void `setLog` (LogPtr &*log*)
Set the stage's log.

Parameters

- *log*: *Log* (page 407) pointer.

virtual LogPtr `log` () const
Return the stage's log pointer.

Return *Log* (page 407) pointer.

void `startLogging` () const
Push the stage's leader into the log.

void `stopLogging` () const
Pop the stage's leader from the log.

bool `isDebug` () const
Determine whether the stage is in debug mode or not.

Return The stage's debug state.

virtual std::string `getName` () const = 0
Return the name of a stage.

Return The stage's name.

```
void setTag (const std::string &tag)  
    Set a specific tag name.
```

```
virtual std::string tag () const  
    Return the tag name of a stage.
```

Return The tag name.

```
std::vector<Stage (page 420) *> &getInputs ()  
    Return a list of the stage's inputs.
```

Return A vector pointers to input stages.

```
MetadataNode (page 409) getMetadata () const  
    Get the stage's metadata node.
```

Return *Stage* (page 420)'s metadata.

```
void serialize (MetadataNode (page 409) root, PipelineWriter::TagMap  
    &tags) const  
    Serialize a stage by inserting appropriate data into the provided MetadataNode  
(page 409).
```

Used to dump a pipeline specification in a portable format.

Parameters

- **root**: Node to which a stage's metadata should be added.
- **tags**: Pipeline writer's current list of stage tags.

Public Static Functions

```
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.

Return Whether the parsed name is a valid stage name.

Parameters

- **o**: Input string to parse.

- `pos`: Parsing start/end position.

`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.

Return Whether the parsed name is a valid tag name.

Parameters

- `o`: Input string to parse.
- `pos`: Parsing start/end position.
- `tag`: Parsed tag name.

`pdal::StageFactory`

`class pdal::StageFactory`

This class provides a mechanism for creating `Stage` (page 420) 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 426) if desired.

Note `Stage` (page 420) creation is thread-safe.

Public Functions

`StageFactory (bool no_plugins = true)`

Create a stage factory.

Parameters

- `no_plugins`: Don't load plugins or allowed them to be created with this factory.

`Stage (page 420) *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.

Return Pointer to created stage.

Parameters

- `stage_name`: Type of stage to be created.

`void destroyStage (Stage *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

`StringList extensions (const std::string &driver)`

Return the file extensions associated with a driver.

Parameters

- `driver`: Name of the driver whose extensions should be returned.

`std::string inferReaderDriver (const std::string &filename)`

Infer the reader to use based on a filename.

Return Driver name or empty string if no reader can be inferred from the filename.

Parameters

- `filename`: Filename that should be analyzed to determine a driver.

`std::string inferWriterDriver (const std::string &filename)`

Infer the writer to use based on filename extension.

Return Driver name or empty string if no writer can be inferred from the filename.

`pdal::Utils`

:cpp:namespace:`'pdal::Utils'` is a set of utility functions.

`namespace pdal::Utils`

Functions

```
template <>
template<>
bool fromString<Eigen::MatrixXd> (const std::string &s, Eigen::MatrixXd
&matrix)

std::string PDAL_DLL pdal::Utils::toJSON(const MetadataNode &m)

void PDAL_DLL pdal::Utils::toJSON(const MetadataNode &m, std::ostream &os)

std::ostream PDAL_DLL * pdal::Utils::createFile(const std::string & path)
Create a file (may be on a supported remote filesystem).
```

Return Pointer to the created stream, or NULL.

Parameters

- path: Path to file to create.
- asBinary: Whether the file should be written in binary mode.

```
std::istream PDAL_DLL * pdal::Utils::openFile(const std::string & path)
Open a file (potentially on a remote filesystem).
```

Return Pointer to stream opened for input.

Parameters

- path: Path (potentially remote) of file to open.
- asBinary: Whether the file should be opened binary.

```
void PDAL_DLL pdal::Utils::closeFile(std::ostream * out)
Close an output stream.
```

Parameters

- out: Stream to close.

```
void PDAL_DLL pdal::Utils::closeFile(std::istream * in)
Close an input stream.
```

Parameters

- out: Stream to close.

```
bool PDAL_DLL pdal::Utils::fileExists(const std::string & path)
Check to see if a file exists.
```

Return Whether the file exists or not.

Parameters

- path: Path to file.

```
double PDAL_DLL pdal::Utils::computeHausdorff(PointViewPtr srcView, PointViewPtr dstView)
void printError(const std::string &s)
double toDouble(const Everything &e, Dimension (page 395)::Type
               (page 395) type)
template <typename INPUT>
Everything extractDim(INPUT &ext, Dimension (page 395)::Type (page 395)
                      type)
template <typename OUTPUT>
void insertDim(OUTPUT &ins, Dimension (page 395)::Type (page 395) type,
               const Everything &e)
MetadataNode (page 409) toMetadata (const BOX2D (page 387) &bounds)
MetadataNode (page 409) toMetadata (const BOX3D (page 390) &bounds)
int openProgress (const std::string &filename)
void closeProgress (int fd)
void writeProgress (int fd, const std::string &type, const std::string &text)
std::vector<std::string> PDAL_DLL pdal::Utils::maybeGlob(const std::string &glob)
template <typename CONTAINER, typename VALUE>
bool contains (const CONTAINER &cont, const VALUE &val)
    Determine if a container contains a value.
```

Return true if the value is in the container, false otherwise.

Parameters

- cont: Container.
- val: Value.

```
template <typename KEY, typename VALUE>
bool contains (const std::map<KEY, VALUE> &c, const KEY &v)
    Determine if a map contains a key.
```

Return true if the value is in the container, false otherwise.

Parameters

- c: Map.
- v: Key value.

```
template <typename CONTAINER, typename VALUE>
```

```
void remove (CONTAINER &cont, const VALUE &val)
```

Remove all instances of a value from a container.

Parameters

- *cont*: Container.
- *v*: Value to remove.

```
template <typename CONTAINER, typename PREDICATE>
```

```
void remove_if (CONTAINER &cont, PREDICATE p)
```

Remove all instances matching a unary predicate from a container.

Parameters

- *cont*: Container.
- *p*: Predicate indicating whether a value should be removed.

```
template <class T>
```

```
PDAL_DLL const T& pdal::Utils::clamp(const T &t, const T &min, const
```

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.

Return the value to clamped to the given bounds.

Parameters

- *t*: the input value.
- *min*: the lower bound.
- *max*: the upper bound.

```
void random_seed (unsigned int seed)
```

Set a seed for random number generation.

Parameters

- *seed*: Seed value.

```
double random (double minimum, double maximum)
```

Generate a random value in the range [minimum, maximum].

Parameters

- *minimum*: Lower value of range for random number generation.
- *maximum*: Upper value of range for random number generation.

```
double uniform (const double &minimum, const double &maximum, uint32_t  
              seed)
```

Generate values in a uniform distribution in the range [*minimum*, *maximum*] using the provided seed value.

Parameters

- *double*: Lower value of range for random number generation.
- *double*: Upper value of range for random number generation.
- *seed*: Seed value for random number generation.

```
double normal (const double &mean, const double &sigma, uint32_t seed)
```

Generate values in a normal distribution in the range [*minimum*, *maximum*] using the provided seed value.

Parameters

- *double*: Lower value of range for random number generation.
- *double*: Upper value of range for random number generation.
- *seed*: Seed value for random number generation.

```
PDAL_DLL bool pdal::Utils::compare_approx(double v1, double v2, double t)
```

Determine if two values are within a particular range of each other.

Parameters

- *v1*: First value to compare.
- *v2*: Second value to compare.
- *tolerance*: Maximum difference between *v1* and *v2*

```
double sround (double r)
```

Round double value to nearest integral value.

Return Rounded value

Parameters

- *r*: Value to round

```
std::string tolower (const std::string &s)
```

Convert a string to lowercase.

Return Converted string.

```
std::string toupper (const std::string &s)
Convert a string to uppercase.
```

Return Converted string.

```
bool iequals (const std::string &s, const std::string &s2)
Compare strings in a case-insensitive manner.
```

Return Whether the strings are equal.

Parameters

- s: First string to compare.
- s2: Second string to compare.

```
bool startsWith (const std::string &s, const std::string &prefix)
Determine if a string starts with a particular prefix.
```

Return Whether the string begins with the prefix.

Parameters

- s: String to check for prefix.
- prefix: Prefix to search for.

```
int cksum (char *buf, size_t size)
Generate a checksum that is the integer sum of the values of the bytes in a buffer.
```

Return Generated checksum.

Parameters

- buf: Pointer to buffer.
- size: Size of buffer.

```
int getenv (std::string const &name, std::string &val)
Fetch the value of an environment variable.
```

Return 0 on success, -1 on failure

Parameters

- name: Name of environment variable.
- name: Value of the environment variable if it exists, empty otherwise.

`int setenv (const std::string &env, const std::string &val)`
Set the value of an environment variable.

Return 0 on success, -1 on failure

Parameters

- *env*: Name of environment variable.
- *val*: Value of environment variable.

`int unsetenv (const std::string &env)`
Clear the value of an environment variable.

Return 0 on success, -1 on failure

Parameters

- *env*: Name of the environment variable to clear.

`void eatwhitespace (std::istream &s)`
Skip stream input until a non-space character is found.

Parameters

- *s*: Stream to process.

`void trimLeading (std::string &s)`
Remove whitespace from the beginning of a string.

Parameters

- *s*: String to be trimmed.

`void trimTrailing (std::string &s)`
Remove whitespace from the end of a string.

Parameters

- *s*: String to be trimmed.

`void trim (std::string &s)`
Remove whitespace from the beginning and end of a string.

Parameters

- *s*: String to be trimmed.

```
bool eatcharacter (std::istream &s, char x)
```

If specified character is at the current stream position, advance the stream position by 1.

Return `true` if the character is at the current stream position, `false` otherwise.

Parameters

- `s`: Stream to inspect.
- `x`: Character to check for.

```
std::string base64_encode (const unsigned char *buf, size_t size)
```

Convert a buffer to a string using base64 encoding.

Return Encoded buffer.

Parameters

- `buf`: Pointer to buffer to encode.
- `size`: Size of buffer.

```
std::string base64_encode (std::vector<uint8_t> const &bytes)
```

Convert a buffer to a string using base64 encoding.

Return Encoded buffer.

Parameters

- `bytes`: Pointer to buffer to encode.

```
std::vector<uint8_t> base64_decode (std::string const &input)
```

Decode a base64-encoded string into a buffer.

Return Buffer containing decoded string.

Parameters

- `input`: String to decode.

```
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.

Return Pointer to FILE for input/output from the subprocess.

Parameters

- command: Command to run in subprocess. Either ‘r’, ‘w’ or ‘r+’ to specify if the pipe should be opened as read-only, write-only or read-write.

```
int portable_pclose (FILE *fp)
```

Close file opened with *portable_popen* (page 433).

Return 0 on success, -1 on failure.

Parameters

- fp: Pointer to file to close.

```
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,

```
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.

Return Modified version of input string.

Parameters

- input: Input string to modify.
- replaceWhat: Token to locate in input string.
- replaceWithWhat: Replacement for found tokens.

```
StringList 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.

Return List of substrings generated from the input string.

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`.

```
StringList 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.

Return List of substrings generated from the input string.

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`.

```
std::string escapeJSON (const std::string &s)
```

Add escape characters or otherwise transform an input string so as to be a valid JSON string.

Return Valid JSON version of input string.

Parameters

- `s`: Input string.

```
std::string demangle (const std::string &s)
```

Demangle a C++ symbol into readable form.

Demangle strings using the compiler-provided demangle function.

Return Demangled symbol.

Return Demangled string

Parameters

- `s`: String to demangle.

Parameters

- `s`: String to be demangled.

```
int screenWidth()
```

Return the screen width of an associated tty.

Return The tty screen width or 80 if on Windows or it can't be determined.

```
std::string escapeNonprinting(const std::string &s)
```

Escape non-printing characters by using standard notation (i.e.
) or hex notation () as necessary.

Return Copy of input string with non-printing characters converted to printable
notation.

Parameters

- `s`: String to modify.

```
double normalizeLongitude(double longitude)
```

Normalize longitude so that it's between (-180, 180].

Return Normalized longitude.

Parameters

- `longitude`: Longitude to normalize.

```
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.

Return Buffer converted to hex string.

Parameters

- `buf`: Point to buffer to dump.
- `count`: Size of buffer.

```
std::vector<std::string> backtrace()
```

Generate a backtrace as a list of strings.

Return List of functions at the point of the call.

```
template <typename PREDICATE>
```

```
PDAL_DLL std::string::size_type pdal::Utils::extract(const std::string
```

Count the number of characters in a string that meet a predicate.

Return Then number of characters matching the 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.

```
PDAL_DLL std::string::size_type pdal::Utils::extractSpaces(const std::string
```

Count the number of characters spaces in a string at a position.

Return Then number of space-y characters matching the predicate.

Parameters

- s: String in which to start counting characters.
- p: Position in input string at which to start counting.

```
template <typename PREDICATE>
```

```
PDAL_DLL std::vector<std::string> pdal::Utils::split(const std::string
```

Split a string into substrings based on a predicate.

Characters matching the predicate are discarded.

Return Substrings.

Parameters

- s: String to split.
- p: Unary predicate that returns true to indicate that a character is a split location.

```
template <typename PREDICATE>
```

```
PDAL_DLL std::vector<std::string> pdal::Utils::split2(const std::string
```

Split a string into substrings.

Characters matching the predicate are discarded, as are empty strings otherwise produced by [split\(\)](#) (page 437).

Return Vector of substrings.

Parameters

- s: String to split.
- p: Predicate returns true if a char in a string is a split location.

```
PDAL_DLL std::vector<std::string> pdal::Utils::split(const std::string &s)
```

Split a string into substrings based a splitting character.

The splitting characters are discarded.

Return Substrings.

Parameters

- *s*: String to split.
- *p*: Character indicating split positions.

```
PDAL_DLL std::vector<std::string> pdal::Utils::split2(const std::string &s)
```

Split a string into substrings based a splitting character.

The splitting characters are discarded as are empty strings otherwise produced by [split\(\)](#) (page 437).

Return Substrings.

Parameters

- *s*: String to split.
- *p*: Character indicating split positions.

```
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.

Return List of arguments.

Parameters

- *s*: Input string to parse.

```
template <typename T>  
std::string typeidName()
```

Return a string representation of a type specified by the template argument.

Return String representation of the type.

[RedirectStream](#) (page 444) **redirect** (std::ostream &*out*, std::ostream &*dst*)

Redirect a stream to some other stream, by default a null stream.

Return Context for stream restoration (see [restore\(\)](#) (page 439)).

Parameters

- `out`: Stream to redirect.
- `dst`: Destination stream.

RedirectStream (page 444) **redirect** (`std::ostream &out`)

Redirect a stream to a null stream.

Return Context for stream restoration (see *restore()* (page 439)).

Parameters

- `out`: Stream to redirect.

RedirectStream (page 444) **redirect** (`std::ostream &out, const std::string &file`)

Redirect a stream to some file.

Return Context for stream restoration (see *restore()* (page 439)).

Parameters

- `out`: Stream to redirect.
- `file`: Name of file where stream should be redirected.

`void restore (std::ostream &out, RedirectStream (page 444) &redir)`

Restore a stream redirected with *redirect()* (page 438).

Parameters

- `out`: Stream to be restored.
- `redir`: *RedirectStream* (page 444) returned from corresponding *redirect()* (page 438) 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.

Return Whether value can be safely converted to template type.

Parameters

- `in`: Value to range test.

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.

Return Whether value can be safely converted to template type.

Parameters

- *in*: Value to range test.

`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.

Return `true` if the conversion was successful, `false` if the datatypes/input value don't allow conversion.

Parameters

- *in*: Value to convert.
- *out*: Converted value.

`template <typename T>`

`std::string toString (const T &from)`

Convert a value to its string representation by writing to a stringstream.

Return String representation.

Parameters

- *from*: Value to convert.

`std::string toString (bool from)`

Convert a bool to a string.

`std::string toString (double from)`

Convert a double to string with a precision of 10 decimal places.

Return String representation of numeric value.

Parameters

- *from*: Value to convert.

`std::string toString (float from)`

Convert a float to string with a precision of 10 decimal places.

Return String representation of numeric value.

Parameters

- `from`: Value to convert.

`std::string toString (long long from)`

Convert a long long int to string.

Return String representation of numeric value.

Parameters

- `from`: Value to convert.

`std::string toString (unsigned long long from)`

Convert an unsigned long long int to string.

Return String representation of numeric value.

Parameters

- `from`: Value to convert.

`std::string toString (long from)`

Convert a long int to string.

Return String representation of numeric value.

Parameters

- `from`: Value to convert.

`std::string toString (unsigned int from)`

Convert an unsigned int to string.

Return String representation of numeric value.

Parameters

- `from`: Value to convert.

`std::string toString (int from)`

Convert an int to string.

Return String representation of numeric value.

Parameters

- `from`: Value to convert.

`std::string toString (unsigned short from)`

Convert an unsigned short to string.

Return String representation of numeric value.

Parameters

- `from`: Value to convert.

`std::string toString (short from)`

Convert a short int to string.

Return String representation of numeric value.

Parameters

- `from`: Value to convert.

`std::string toString (char from)`

Convert a char (treated as numeric) to string.

Return String representation of numeric value.

Parameters

- `from`: Value to convert.

`std::string toString (unsigned char from)`

Convert an unsigned char (treated as numeric) to string.

Return String representation of numeric value.

Parameters

- `from`: Value to convert.

`std::string toString (signed char from)`

Convert a signed char (treated as numeric) to string.

Return String representation of numeric value.

Parameters

- `from`: Value to convert.

```
template <typename T>
bool fromString(const std::string &from, T *&to)

template <typename T>
bool fromString(const std::string &from, T &to)
```

Convert a string to a value by reading from a string stream.

Return true if the conversion was successful, false otherwise.

Parameters

- from: String to convert.
- to: Converted value.

```
template <>
bool fromString(const std::string &from, std::string &to)

template <>
template<>
bool fromString<char>(const std::string &s, char &to)
```

Convert a numeric string to a char numeric value.

s String to convert.

Return true if the conversion was successful, false otherwise.

Parameters

- to: Converted numeric value.

```
template <>
template<>
bool fromString<unsigned char>(const std::string &s, unsigned char &to)
```

Convert a numeric string to an unsigned char numeric value.

s String to convert.

Return true if the conversion was successful, false otherwise.

Parameters

- to: Converted numeric value.

```
template <>
template<>
bool fromString<signed char>(const std::string &s, signed char &to)
```

Convert a numeric string to a signed char numeric value.

s String to convert.

Return true if the conversion was successful, false otherwise.

Parameters

- to: Converted numeric value.

```
template <>
template<>
bool fromString<double> (const std::string &s, double &d)
    Specialization conversion from string to double to handle Nan.
```

Return true if the conversion was successful, false otherwise.

Parameters

- s: String to be converted.
- d: Converted value.

```
template <typename E>
std::underlying_type<E>::type toNative (E e)
    Return the argument cast to its underlying type.
```

Typically used on an enum.

Return Converted variable.

Parameters

- e: Variable for which to find the underlying type.

```
struct RedirectStream
#include <Utils.hpp>
```

Public Functions

RedirectStream()

Public Members

```
std::ofstream *m_out
std::streambuf *m_buf
std::unique_ptr<NullOStream> m_null
```

pdal::Writer

class pdal::Writer

A [Writer](#) (page 444) 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.

Inherits from [pdal::Stage](#) (page 420)

Subclassed by pdal::DbWriter, pdal::FlexWriter, pdal::GeoWaveWriter, pdal::MatlabWriter, pdal::NullWriter, pdal::PcdWriter, pdal::PlyWriter, pdal::SbetWriter, pdal::TextWriter

12.2.2 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>
#include <pdal/LasHeader.hpp>
#include <pdal/Options.hpp>
```

Initial steps

Opening the dataset in libLAS:

```
LASReaderH LAS_reader;
LASHeaderH LAS_header;
LASSRSRH 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:

```
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:

```
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:

```
try {
    /* actual code */
} catch {
    /* fail in your own way */
}
```

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:

```
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:

```
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:

```
unsigned int n_features = LASHeader_GetPointRecordsCount(LAS_header);
```

is just point count in PDAL:

```
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:

```
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:

```
x = LASPoint_GetX(LAS_point);  
y = LASPoint_GetY(LAS_point);  
z = LASPoint_GetZ(LAS_point);
```

In PDAL, point coordinates are one of the dimensions:

```
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:

```
int return_no = LASPoint_GetReturnNumber(LAS_point);
int n_returns = LASPoint_GetNumberOfReturns(LAS_point);
```

PDAL:

```
int return_no = point_view->getFieldAs<int>(Id::ReturnNumber, idx);
int n_returns = point_view->getFieldAs<int>(Id::NumberOfReturns,
    ↪idx);
```

Classes

libLAS:

```
int point_class = (int) LASPoint_GetClassification(LAS_point);
```

PDAL:

```
int point_class = point_view->getFieldAs<int>(Id::Classification,
    ↪idx);
```

Color

libLAS:

```
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:

```
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:

```
double time = LASPoint_GetTime(LAS_point);
```

PDAL:

```
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.

12.3 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_PCL=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.

- 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 can also use PCL through the [*filters.pclblock*](#) (page 156) mechanism. PDAL also supports reading and writing PCL PCD files using [*readers.pcd*](#) (page 67) and [*writers.pcd*](#) (page 97).

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.

- Are there any command line tools in PDAL similar to LAStools?

Yes. The `pdal` 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 23) describes application operations you can achieve with PDAL.

- Is there any compatibility with libLAS's LAS Utility Applications or LAStools?

No. 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 23)).

12.4 License

Unless otherwise indicated, all files in the PDAL distribution are

Copyright (c) 2017, 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.

12.4.1 Overall PDAL license (BSD)

Copyright (c) 2017, 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.

12.5 References

CHAPTER
THIRTEEN

INDICES AND TABLES

- genindex
- search

BIBLIOGRAPHY

- [MS10] Andriy Myronenko and Xubo Song. Point set registration: coherent point drift. *IEEE transactions on pattern analysis and machine intelligence*, 32(12):2262–75, dec 2010.
- [YG88] Alan L. Yuille and Norberto M. Grzywacz. The Motion Coherence Theory. *Second International Conference on Computer Vision*, 1988.
- [Li2010] Li, Ruosi, et al. “Polygonizing extremal surfaces with manifold guarantees.” Proceedings of the 14th ACM Symposium on Solid and Physical Modeling. ACM, 2010.
- [Breunig2000] Breunig, M.M., Kriegel, H.-P., Ng, R.T., Sander, J., 2000. LOF: Identifying Density-Based Local Outliers. Proc. 2000 Acm Sigmod Int. Conf. Manag. Data 1–12.
- [Mongus2012] Mongus, D., Zalik, B., 2012. Parameter-free ground filtering of LiDAR data for automatic DTM generation. *ISPRS J. Photogramm. Remote Sens.* 67, 1–12.
- [Alexa2003] Alexa, Marc, et al. “Computing and rendering point set surfaces.” *Visualization and Computer Graphics, IEEE Transactions on* 9.1 (2003): 3-15.
- [Kazhdan2006] Kazhdan, Michael, Matthew Bolitho, and Hugues Hoppe. “Poisson surface reconstruction.” Proceedings of the fourth Eurographics symposium on Geometry processing. Vol. 7. 2006.
- [Pingel2013] Pingel, T.J., Clarke, K.C., McBride, W.A., 2013. An improved simple morphological filter for the terrain classification of airborne LiDAR data. *ISPRS J. Photogramm. Remote Sens.* 77, 21–30.
- [Gle07] Craig L. Glennie. Rigorous 3D error analysis of kinematic scanning LiDAR systems. *Journal of Applied Geodesy*, jan 2007.
- [Chen2012] Chen, Ziyue et al. “Upward-Fusion Urban DTM Generating Method Using Airborne Lidar Data.” *ISPRS Journal of Photogrammetry and Remote Sensing* 72 (2012): 121–130.
- [Cook1986] Cook, Robert L. “Stochastic sampling in computer graphics.” *ACM Transactions on Graphics (TOG)* 5.1 (1986): 51-72.
- [Dippe1985] Dippé, Mark AZ, and Erling Henry Wold. “Antialiasing through stochastic sampling.” *ACM Siggraph Computer Graphics* 19.3 (1985): 69-78.

[Mesh2009] ALoopingIcon. “Meshing Point Clouds.” *MESHLAB STUFF*. n.p., 7 Sept. 2009. Web. 13 Nov. 2015.

[Rusu2008] Rusu, Radu Bogdan, et al. “Towards 3D point cloud based object maps for household environments.” *Robotics and Autonomous Systems* 56.11 (2008): 927-941.

[Zhang2003] Zhang, Keqi, et al. “A progressive morphological filter for removing nonground measurements from airborne LIDAR data.” *Geoscience and Remote Sensing, IEEE Transactions on* 41.4 (2003): 872-882.

INDEX

A

Apps, 11

B

boundary, 264

C

capstone, 315

classification, 290

classifications, 279

Clipping, 268

CloudCompare, 8

Colorization, 274

Command line, 11

coordinate system, 250

csd, 313

CSV, 249

D

Denoising, 278

Density, 282

density, 287

Docker, 15

DSM, 299

DTM, 299

E

elevation model, 299

Embed, 189

Entwine, 8

Extension, 190

F

filtering, 290

Fusion, 8

G

GDAL, 274

georeferencing, 242, 313

GeoWave, 54, 84

GNSS/IMU, 242, 313

Greyhound, 8

greyhound, 261

ground, 290

H

hexagon tessellation, 282

histogram, 313

I

info command, 248

Install, 191

installation, 248

J

JSON, 249

L

LASTools, 7

libLAS, 8

M

matplotlib, 313

metadata, 250

N

nearby, 253

nearest, 253

Numpy, 11, 189, 313

O

OGR, 264, 268, 282

Optech, 313
OrfeoToolbox, 8
OSGeo4W, 246
outliers, 278

P

PCL, 7, 451
pdal::BOX2D (C++ class), 387
pdal::BOX2D::BOX2D (C++ function), 387
pdal::BOX2D::clear (C++ function), 387
pdal::BOX2D::clip (C++ function), 389
pdal::BOX2D::contains (C++ function), 388, 389
pdal::BOX2D::empty (C++ function), 387
pdal::BOX2D::equal (C++ function), 388
pdal::BOX2D::getDefaultSpatialExtent (C++ function), 390
pdal::BOX2D::grow (C++ function), 387, 389
pdal::BOX2D::maxx (C++ member), 390
pdal::BOX2D::maxy (C++ member), 390
pdal::BOX2D::minx (C++ member), 390
pdal::BOX2D::miny (C++ member), 390
pdal::BOX2D::operator = (C++ function), 388
pdal::BOX2D::operator== (C++ function), 388
pdal::BOX2D::overlaps (C++ function), 389
pdal::BOX2D::toBox (C++ function), 389
pdal::BOX2D::toGeoJSON (C++ function), 390
pdal::BOX2D::toWKT (C++ function), 389
pdal::BOX2D::valid (C++ function), 387
pdal::BOX3D (C++ class), 390
pdal::BOX3D::BOX3D (C++ function), 390, 391
pdal::BOX3D::clear (C++ function), 391
pdal::BOX3D::clip (C++ function), 393
pdal::BOX3D::contains (C++ function), 391, 392
pdal::BOX3D::empty (C++ function), 391
pdal::BOX3D::equal (C++ function), 392
pdal::BOX3D::getDefaultSpatialExtent (C++ function), 394
pdal::BOX3D::grow (C++ function), 391, 393
pdal::BOX3D::maxz (C++ member), 394

pdal::BOX3D::minz (C++ member), 394
pdal::BOX3D::operator = (C++ function), 392
pdal::BOX3D::operator== (C++ function), 392
pdal::BOX3D::overlaps (C++ function), 393
pdal::BOX3D::to2d (C++ function), 393
pdal::BOX3D::toBox (C++ function), 393
pdal::BOX3D::toWKT (C++ function), 393
pdal::BOX3D::valid (C++ function), 391
pdal::Charbuf (C++ class), 394
pdal::Charbuf::Charbuf (C++ function), 394
pdal::Charbuf::initialize (C++ function), 395
pdal::Dimension (C++ type), 395
pdal::Dimension::base (C++ function), 396
pdal::Dimension::BaseType (C++ type), 395
pdal::Dimension::COUNT (C++ member), 397
pdal::Dimension::DetailList (C++ type), 395
pdal::Dimension::Double (C++ enumerator), 396
pdal::Dimension::extractName (C++ function), 396
pdal::Dimension::Float (C++ enumerator), 396
pdal::Dimension::Floating (C++ enumerator), 395
pdal::Dimension::fromName (C++ function), 396
pdal::Dimension::interpretationName (C++ function), 396
pdal::Dimension::None (C++ enumerator), 395
pdal::Dimension::operator>> (C++ function), 397
pdal::Dimension::operator<< (C++ function), 397
pdal::Dimension::PROPRIETARY (C++ member), 397
pdal::Dimension::Signed (C++ enumerator), 395
pdal::Dimension::Signed16 (C++ enumerator), 395
pdal::Dimension::Signed32 (C++ enumerator), 396

pdal::Dimension::Signed64 (C++ enumerator), 396
pdal::Dimension::Signed8 (C++ enumerator), 395
pdal::Dimension::size (C++ function), 396
pdal::Dimension::toName (C++ function), 396
pdal::Dimension::type (C++ function), 396
pdal::Dimension::Type (C++ type), 395
pdal::Dimension::Unsigned (C++ enumerator), 395
pdal::Dimension::Unsigned16 (C++ enumerator), 395
pdal::Dimension::Unsigned32 (C++ enumerator), 396
pdal::Dimension::Unsigned64 (C++ enumerator), 396
pdal::Dimension::Unsigned8 (C++ enumerator), 395
pdal::Extractor (C++ class), 397
pdal::Extractor::Extractor (C++ function), 397
pdal::Extractor::get (C++ function), 398, 399
pdal::Extractor::good (C++ function), 398
pdal::Extractor::operator bool (C++ function), 397
pdal::Extractor::operator>> (C++ function), 399
pdal::Extractor::position (C++ function), 398
pdal::Extractor::seek (C++ function), 397
pdal::Extractor::skip (C++ function), 398
pdal::FileUtils (C++ type), 399
pdal::Filter (C++ class), 404
pdal::Filter::Filter (C++ function), 404
pdal::IStream (C++ class), 404
pdal::IStream::~IStream (C++ function), 405
pdal::IStream::IStream (C++ function), 404, 405
pdal::IStream::operator bool (C++ function), 405
pdal::Log (C++ class), 407
pdal::Log::~Log (C++ function), 407
pdal::Log::clearFloat (C++ function), 408
pdal::Log::floatPrecision (C++ function), 408
pdal::Log::get (C++ function), 408
pdal::Log::getLevel (C++ function), 407
pdal::Log::getLevelString (C++ function), 408
pdal::Log::getLogStream (C++ function), 408
pdal::Log::leader (C++ function), 408
pdal::Log::Log (C++ function), 409
pdal::Log::popLeader (C++ function), 408
pdal::Log::pushLeader (C++ function), 408
pdal::Log::setLeader (C++ function), 408
pdal::Log::setLevel (C++ function), 407
pdal::Metadata (C++ class), 409
pdal::Metadata::getNode (C++ function), 409
pdal::Metadata::inferType (C++ function), 409
pdal::Metadata::Metadata (C++ function), 409
pdal::MetadataNode (C++ class), 409
pdal::MetadataNode::add (C++ function), 410
pdal::MetadataNode::addEncoded (C++ function), 410
pdal::MetadataNode::addList (C++ function), 410
pdal::MetadataNode::addListEncoded (C++ function), 410
pdal::MetadataNode::addOrUpdate (C++ function), 410
pdal::MetadataNode::addWithType (C++ function), 410
pdal::MetadataNode::childNames (C++ function), 411
pdal::MetadataNode::children (C++ function), 411
pdal::MetadataNode::clone (C++ function), 410
pdal::MetadataNode::description (C++ function), 411
pdal::MetadataNode::empty (C++ function), 411
pdal::MetadataNode::find (C++ function), 411
pdal::MetadataNode::findChild (C++ function), 411
pdal::MetadataNode::findChildren (C++ function), 411
pdal::MetadataNode::hasChildren (C++ function), 411
pdal::MetadataNode::jsonValue (C++ function), 411
pdal::MetadataNode::kind (C++ function),

410
pdal::MetadataNode::MetadataNode (C++ function), 410
pdal::MetadataNode::name (C++ function), 410
pdal::MetadataNode::operator (C++ function), 411
pdal::MetadataNode::type (C++ function), 410
pdal::MetadataNode::valid (C++ function), 411
pdal::MetadataNode::value (C++ function), 410, 411
pdal::Options (C++ class), 411
pdal::Options::add (C++ function), 411, 412
pdal::Options::addConditional (C++ function), 411
pdal::Options::getKeys (C++ function), 412
pdal::Options::getOptions (C++ function), 412
pdal::Options::getValues (C++ function), 412
pdal::Options::Options (C++ function), 411
pdal::Options::remove (C++ function), 412
pdal::Options::replace (C++ function), 412
pdal::Options::toCommandLine (C++ function), 412
pdal::Options::toMetadata (C++ function), 412
pdal::PointTable (C++ class), 412
pdal::PointTable::~PointTable (C++ function), 412
pdal::PointTable::PointTable (C++ function), 412
pdal::PointTable::supportsView (C++ function), 412
pdal::PointView (C++ class), 413
pdal::PointView::~PointView (C++ function), 413
pdal::PointView::append (C++ function), 413
pdal::PointView::appendPoint (C++ function), 413
pdal::PointView::begin (C++ function), 413
pdal::PointView::build2dIndex (C++ function), 415
pdal::PointView::build3dIndex (C++ function), 415
pdal::PointView::calculateBounds (C++ function), 414, 416
pdal::PointView::clearTemps (C++ function), 415
pdal::PointView::compare (C++ function), 413
pdal::PointView::createMesh (C++ function), 415
pdal::PointView::dimName (C++ function), 414
pdal::PointView::dims (C++ function), 414
pdal::PointView::dimSize (C++ function), 414
pdal::PointView::dimType (C++ function), 414
pdal::PointView::dimTypes (C++ function), 414
pdal::PointView::dump (C++ function), 414
pdal::PointView::empty (C++ function), 413
pdal::PointView::end (C++ function), 413
pdal::PointView::getField (C++ function), 413
pdal::PointView::getFieldAs (C++ function), 413
pdal::PointView::getOrAddPoint (C++ function), 415
pdal::PointView::getPackedPoint (C++ function), 414
pdal::PointView::getPoint (C++ function), 415
pdal::PointView::getRawField (C++ function), 414
pdal::PointView::hasDim (C++ function), 414
pdal::PointView::id (C++ function), 413
pdal::PointView::layout (C++ function), 414
pdal::PointView::makeNew (C++ function), 413
pdal::PointView::mesh (C++ function), 415
pdal::PointView::operator= (C++ function), 413
pdal::PointView::point (C++ function), 413
pdal::PointView::pointSize (C++ function), 414
pdal::PointView::PointView (C++ function), 413
pdal::PointView::setField (C++ function), 413
pdal::PointView::setPackedPoint (C++ function), 413

function), 414
pdal::PointView::setSpatialReference (C++ function), 414
pdal::PointView::size (C++ function), 413
pdal::PointView::spatialReference (C++ function), 414
pdal::PointView::toMetadata (C++ function), 415
pdal::ProgramArgs (C++ class), 416
pdal::ProgramArgs::add (C++ function), 416–418
pdal::ProgramArgs::addSynonym (C++ function), 418
pdal::ProgramArgs::commandLine (C++ function), 419
pdal::ProgramArgs::dump (C++ function), 419
pdal::ProgramArgs::dump2 (C++ function), 419
pdal::ProgramArgs::dump3 (C++ function), 419
pdal::ProgramArgs::parse (C++ function), 418
pdal::ProgramArgs::parseSimple (C++ function), 418
pdal::ProgramArgs::reset (C++ function), 419
pdal::ProgramArgs::set (C++ function), 417
pdal::Reader (C++ class), 420
pdal::Stage (C++ class), 420
pdal::Stage::~Stage (C++ function), 420
pdal::Stage::addAllArgs (C++ function), 422
pdal::Stage::addConditionalOptions (C++ function), 422
pdal::Stage::addOptions (C++ function), 423
pdal::Stage::execute (C++ function), 421
pdal::Stage::getInputs (C++ function), 424
pdal::Stage::getMetadata (C++ function), 424
pdal::Stage::getName (C++ function), 423
pdal::Stage::getSpatialReference (C++ function), 422
pdal::Stage::isDebug (C++ function), 423
pdal::Stage::log (C++ function), 423
pdal::Stage::parseName (C++ function), 424
pdal::Stage::parseTagName (C++ function), 425
pdal::Stage::prepare (C++ function), 421
pdal::Stage::preview (C++ function), 421
pdal::Stage::removeOptions (C++ function), 423
pdal::Stage::serialize (C++ function), 424
pdal::Stage::setInput (C++ function), 420
pdal::Stage::setLog (C++ function), 423
pdal::Stage::setOptions (C++ function), 422
pdal::Stage::setProgressFd (C++ function), 421
pdal::Stage::setSpatialReference (C++ function), 422
pdal::Stage::setTag (C++ function), 424
pdal::Stage::Stage (C++ function), 420
pdal::Stage::startLogging (C++ function), 423
pdal::Stage::stopLogging (C++ function), 423
pdal::Stage::tag (C++ function), 424
pdal::StageFactory (C++ class), 425
pdal::StageFactory::createStage (C++ function), 425
pdal::StageFactory::destroyStage (C++ function), 426
pdal::StageFactory::extensions (C++ function), 426
pdal::StageFactory::inferReaderDriver (C++ function), 426
pdal::StageFactory::inferWriterDriver (C++ function), 426
pdal::StageFactory::StageFactory (C++ function), 425
pdal::Utils (C++ type), 426
pdal::Utils::backtrace (C++ function), 436
pdal::Utils::base64_decode (C++ function), 433
pdal::Utils::base64_encode (C++ function), 433
pdal::Utils::cksum (C++ function), 431
pdal::Utils::closeProgress (C++ function), 428
pdal::Utils::contains (C++ function), 428
pdal::Utils::demangle (C++ function), 435
pdal::Utils::eatcharacter (C++ function), 432
pdal::Utils::eatwhiteSpace (C++ function), 432
pdal::Utils::escapeJSON (C++ function), 435
pdal::Utils::escapeNonprinting (C++ function), 436

pdal::Utils::extractDim (C++ function), 428
pdal::Utils::fromString (C++ function), 442, 443
pdal::Utils::fromString<char> (C++ function), 443
pdal::Utils::fromString<double> (C++ function), 443
pdal::Utils::fromString<Eigen::MatrixXd> (C++ function), 427
pdal::Utils::fromString<signed char> (C++ function), 443
pdal::Utils::fromString<unsigned char> (C++ function), 443
pdal::Utils::getenv (C++ function), 431
pdal::Utils::hexDump (C++ function), 436
pdal::Utils::iequals (C++ function), 431
pdal::Utils::inRange (C++ function), 439
pdal::Utils::insertDim (C++ function), 428
pdal::Utils::normal (C++ function), 430
pdal::Utils::normalizeLongitude (C++ function), 436
pdal::Utils::numericCast (C++ function), 440
pdal::Utils::openProgress (C++ function), 428
pdal::Utils::portable_pclose (C++ function), 434
pdal::Utils::portable_popen (C++ function), 433
pdal::Utils::printError (C++ function), 428
pdal::Utils::random (C++ function), 429
pdal::Utils::random_seed (C++ function), 429
pdal::Utils::redirect (C++ function), 438, 439
pdal::Utils::RedirectStream (C++ class), 444
pdal::Utils::RedirectStream::m_buf (C++ member), 444
pdal::Utils::RedirectStream::m_null (C++ member), 444
pdal::Utils::RedirectStream::m_out (C++ member), 444
pdal::Utils::RedirectStream::RedirectStream (C++ function), 444
pdal::Utils::remove (C++ function), 428
pdal::Utils::remove_if (C++ function), 429
pdal::Utils::replaceAll (C++ function), 434
pdal::Utils::restore (C++ function), 439
pdal::Utils::run_shell_command (C++ function), 434
pdal::Utils::screenWidth (C++ function), 436
pdal::Utils::setenv (C++ function), 431
pdal::Utils::simpleWordexp (C++ function), 438
pdal::Utils::round (C++ function), 430
pdal::Utils::startsWith (C++ function), 431
pdal::Utils::toDouble (C++ function), 428
pdal::Utils::tolower (C++ function), 430
pdal::Utils::toMetadata (C++ function), 428
pdal::Utils::toNative (C++ function), 444
pdal::Utils::toString (C++ function), 440–442
pdal::Utils::toupper (C++ function), 430
pdal::Utils::trim (C++ function), 432
pdal::Utils::trimLeading (C++ function), 432
pdal::Utils::trimTrailing (C++ function), 432
pdal::Utils::typeidName (C++ function), 438
pdal::Utils::uniform (C++ function), 430
pdal::Utils::unsetenv (C++ function), 432
pdal::Utils::wordWrap (C++ function), 434
pdal::Utils::wordWrap2 (C++ function), 435
pdal::Utils::writeProgress (C++ function), 428
pdal::Writer (C++ class), 444
poisson, 287
Potree, 261
project, 315
pronounce, 451
Python, 11, 189–191, 313

Q

QGIS, 264
query, 253
Quickstart, 15

R

range filter, 279
Raster, 274
References, 453
Reprojection, 256
RGB, 274
Riegl, 313

S

sample, 287
search, 253

SOCS, [242](#)

software installation, [246](#)

spatial reference system, [250](#)

Stage, [420](#)

Start Here, [248](#)

T

thinning, [287](#)

U

Utils, [426](#)

UTM, [256](#), [313](#)

V

Vector, [268](#)

voxel sampling, [287](#)

W

web services, [261](#)

WGS84, [256](#), [313](#)