



PDAL: Point cloud Data Abstraction Library

1.4.0

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

Dec 15, 2016

CONTENTS

1	News	3
1.1	08-29-2016	3
2	Download	5
2.1	Download	5
3	Quickstart	9
3.1	Quickstart	9
4	Applications	17
4.1	Applications	17
5	Community	35
5.1	Community	35
6	Drivers	37
6.1	Pipeline	37
6.2	Readers	48
6.3	Writers	74
6.4	Filters	101
6.5	Dimensions	161
7	Tutorials	165
7.1	Tutorials	165
8	Workshop	265
8.1	Point Cloud Processing and Analysis with PDAL	265

9 Development	365
9.1 Development	365
9.2 API	421
9.3 FAQ	425
9.4 License	426
9.5 References	427
10 Indices and tables	429
Bibliography	431
Index	433



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. See [Readers](#) (page 48) and [Writers](#) (page 74) for data formats PDAL supports, and see [Filters](#) (page 101) 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. See [Applications](#) (page 17) for more information.

The entire website is available as a single PDF at <http://pdal.io/PDAL.pdf>

**CHAPTER
ONE**

NEWS

08-29-2016

PDAL 1.3.0 has been released. Visit [Download](#) (page 5) to obtain a copy of the source code, or follow the [Quickstart](#) (page 9) to get going in a hurry with [Docker](#) (<https://www.docker.com/>).

CHAPTER TWO

DOWNLOAD

Download

Contents

- *Download* (page 5)
 - *Current Release(s)* (page 5)
 - *Past Releases* (page 6)
 - *Development Source* (page 6)
 - *Binaries* (page 6)
 - * *Docker* (page 6)
 - * *Windows* (page 6)
 - * *RPMs* (page 7)
 - * *Debian* (page 7)

Current Release(s)

- **2016-12-15 PDAL-1.4.0-src.tar.gz**
(<http://download.osgeo.org/pdal/PDAL-1.4.0-src.tar.gz>) Release Notes

(<https://github.com/PDAL/PDAL/releases/tag/1.4.0>) ([md5](#)
(<http://download.osgeo.org/pdal/PDAL-1.4.0-src.tar.gz.md5>))

Past Releases

- **2016-08-29** [PDAL-1.3.0-src.tar.gz](#)
(<http://download.osgeo.org/pdal/PDAL-1.3.0-src.tar.gz>) [Release Notes](#)
(<https://github.com/PDAL/PDAL/releases/tag/1.4.0>)
- **2016-03-31** [PDAL-1.2.0-src.tar.gz](#)
(<http://download.osgeo.org/pdal/PDAL-1.2.0-src.tar.gz>) [Release Notes](#)
(<https://github.com/PDAL/PDAL/releases/tag/1.4.0>)

Development Source

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

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

```
git clone git@github.com:PDAL/PDAL.git pdal
```

Binaries

Docker

The fastest way to get going with PDAL is to use the Docker build. See the tutorial at [Install Docker](#) (page 9) for more information.

```
docker pull pdal/pdal:1.2
```

Windows

A 1.1.0 release of PDAL is available via [OSGeo4W](#) (<http://trac.osgeo.org/osgeo4w/>). It is only 64-bit at this time. Use the [Install Docker](#) (page 9) builds if you want to use the PDAL

Applications (page 17), otherwise, a call for help with building current Windows PDAL builds is at <https://lists.osgeo.org/pipermail/pdal/2016-November/001089.html>

RPMs

RPMs for PDAL are available at <http://pdal.s3-website-us-east-1.amazonaws.com/rpms/>

Debian

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

CHAPTER THREE

QUICKSTART

Quickstart

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 9). 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 57) file. It will utilize the PDAL *command line application* (page 17) to inspect the file.

Install Docker

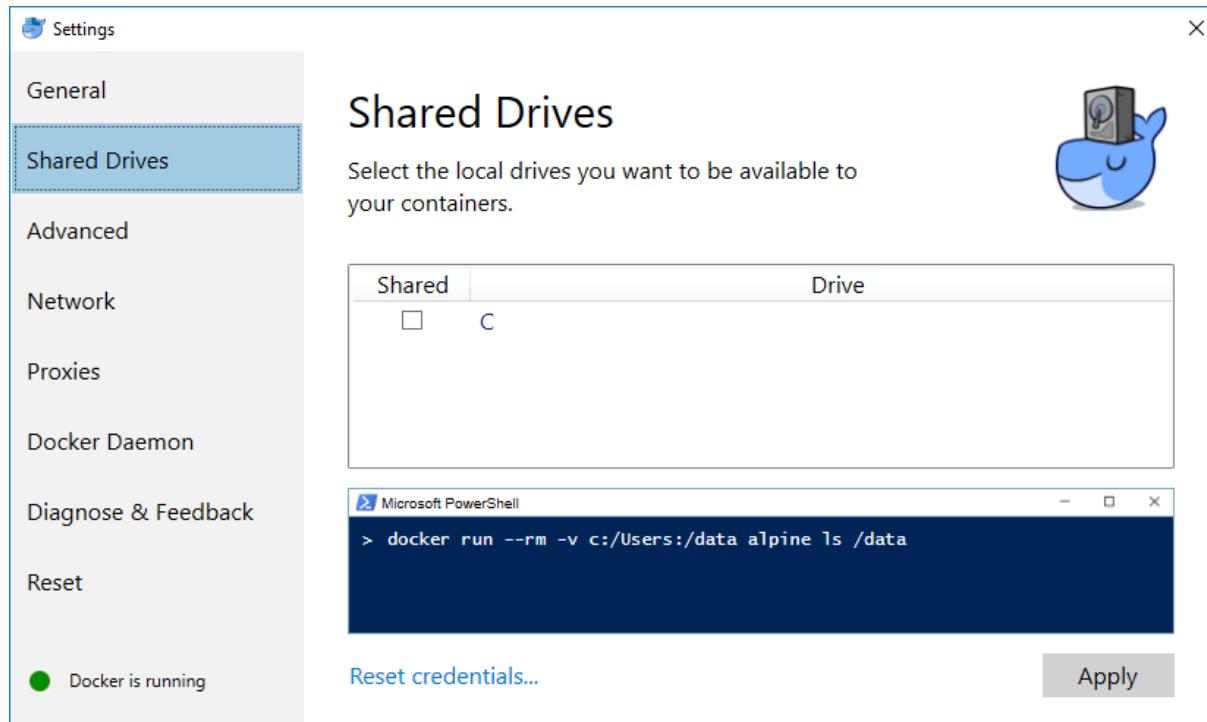
Docker starting documentation can be found at the following links. Read through them a bit for your platform so you have an idea what to expect.

- [Windows](https://docs.docker.com/docker-for-windows/) (<https://docs.docker.com/docker-for-windows/>)
- [OSX](https://docs.docker.com/docker-for-mac/) (<https://docs.docker.com/docker-for-mac/>)
- [Linux](https://docs.docker.com/engine/installation/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 9) 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:3AI5: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:\>
```

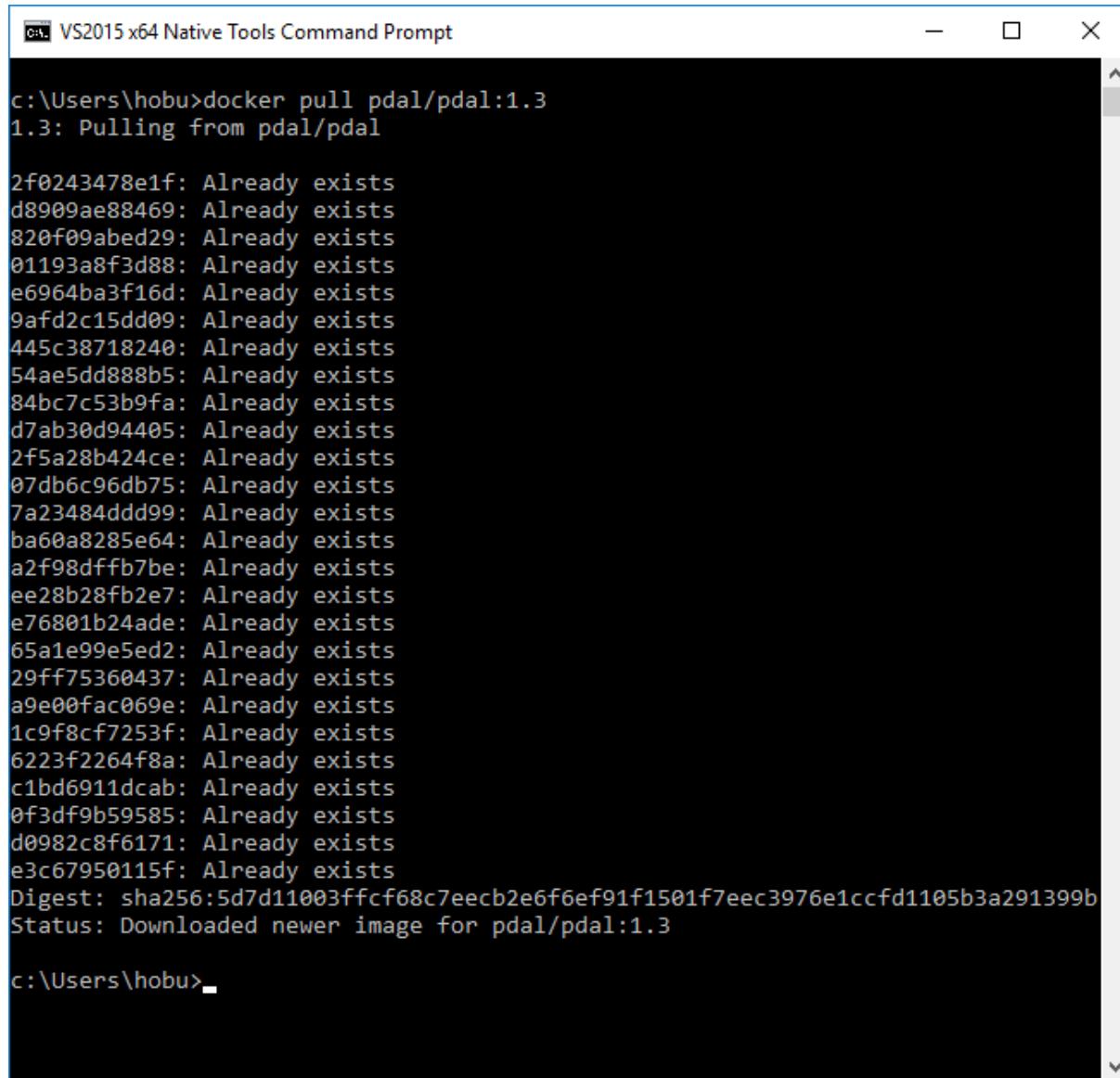
Obtain PDAL Image

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. We

PDAL: Point cloud Data Abstraction Library, 1.4.0

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:1.4
```



The screenshot shows a terminal window titled "VS2015 x64 Native Tools Command Prompt". The command entered is "docker pull pdal/pdal:1.4". The output indicates that the image already exists for every tag listed, and a newer version is downloaded for the tag "1.4".

```
c:\Users\hobu>docker pull pdal/pdal:1.4
1.4: Pulling from pdal/pdal
2f0243478e1f: Already exists
d8909ae88469: Already exists
820f09abed29: Already exists
01193a8f3d88: Already exists
e6964ba3f16d: Already exists
9afdb2c15dd09: 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:1.4

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:1.4, 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.

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.liblas.org/samples/autzen/autzen.laz
```

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

Print the first point

```
docker run -v /c/Users/hobu:/data pdal/pdal:1.4 pdal info /data/  
→autzen.laz -p 0
```

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

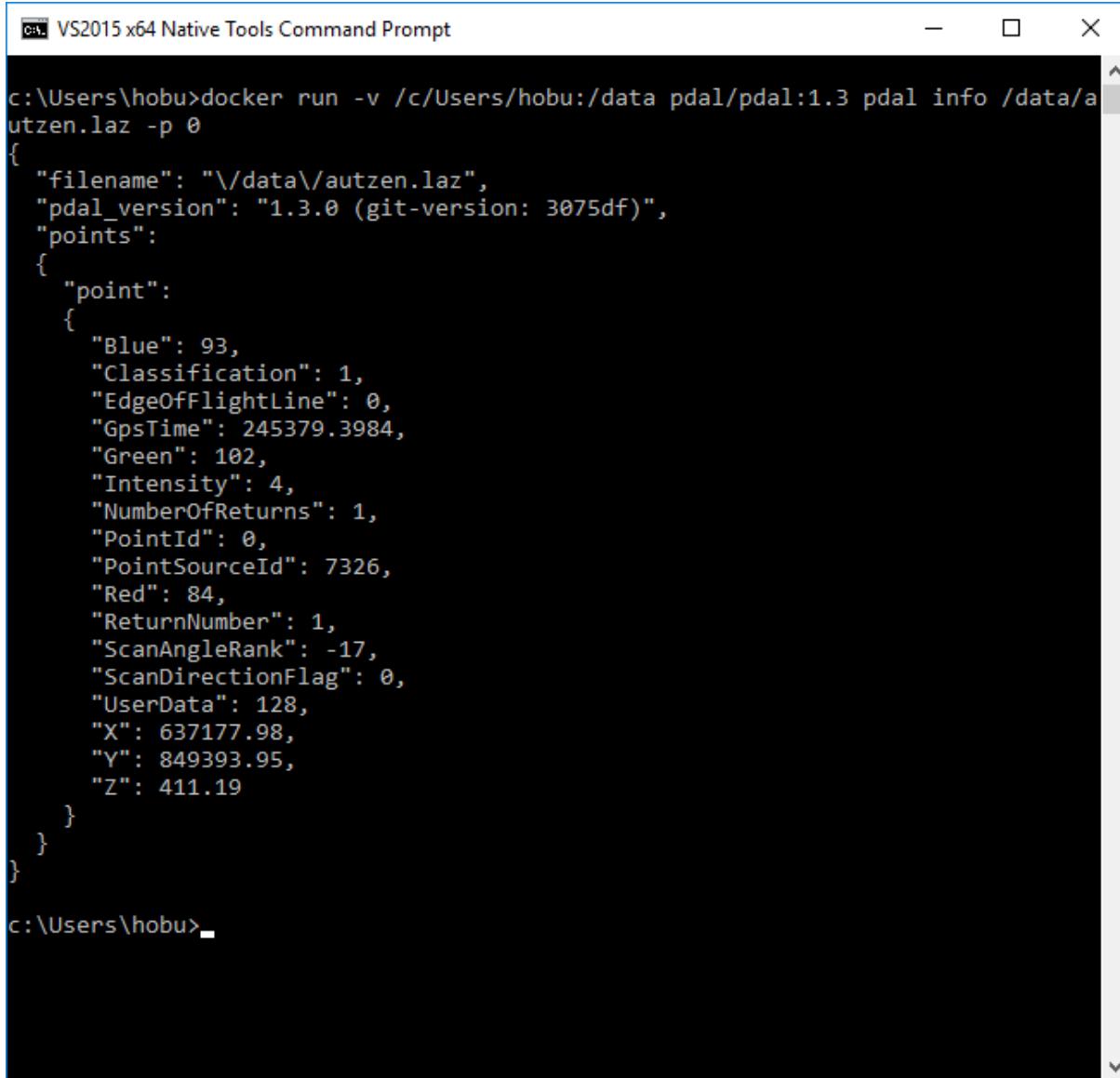
1. `docker`: We are running PDAL within the context of docker, so all of our commands will start with the `docker` command.
2. `run`: Tells docker we're going to run an image
3. `-v /c/Users/hobu:/data`: Maps our home directory to a directory called `/data` inside the container.

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:1.4`: 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 22) 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.



```
VS2015 x64 Native Tools Command Prompt
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>
```

What's next?

- Visit [Applications](#) (page 17) to find out how to utilize PDAL applications to process data on the command line yourself.
- Visit [Development](#) (page 365) to learn how to embed and use PDAL in your own applications.
- [Readers](#) (page 48) lists the formats that PDAL can read, [Filters](#) (page 101) lists the kinds of operations you can do with PDAL, and [Writers](#) (page 74) lists the formats PDAL can write.
- [Tutorials](#) (page 165) contains a number of walk-through tutorials for achieving many tasks with PDAL.
- [The PDAL workshop](#) (page 265) contains numerous hands-on examples with screenshots and example data of how to use PDAL [Applications](#) (page 17) to tackle point cloud data processing tasks.

See also:

[Community](#) (page 35) is a good source to reach out to when you're stuck.

CHAPTER FOUR

APPLICATIONS

Applications

PDAL contains consists of a single application, called `pdal`. Applications are run by invoking the `pdal` application along with the 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
```

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 info --options`

invocation.

delta

The `delta` command is used to select a nearest point from a candidate file for each point in the source file. If the `--2d` option is used, the query only happens in XY coordinate space.

```
$ pdal delta <source> <candidate> [output]
```

Standard out is used if no output file is specified.

```
--source arg      Non-positional option for specifying source filename  
--candidate arg   Non-positional option for specifying candidate filename  
--output arg      Non-positional option for specifying output filename [/dev/stdout]  
--2d              only 2D comparisons/indexing
```

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
```

density

The density command produces a tessellated hexagonal OGR layer from the output of [filters.hexbin](#) (page 123).

Note: The density command is only available when PDAL is linked with Hexer.

```
--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
```

diff

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

```
$ pdal diff <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 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

ground

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

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

```
--developer-debug      Enable developer debug (don't trap exceptions)
--label                 A string to label the process with
--visualize             Visualize result
--driver                Override reader driver
--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/](http://pdal.io/)

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.

info

Dumps 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 arg      Non-positional argument to specify input filename.  
--point [-p] arg Display points for particular points. Points can  
  ↪be specified in  
          a range or list: 4-10, 15, 255-300.  
--query arg     Add a listing of points based on the distance from  
  ↪the provided  
          location. The number of points returned can be  
  ↪limited by  
          providing an optional count.  
          --query "25.34,35.123/3" or --query "11532.23 -  
  ↪10e23 1.234/10"  
--stats         Display the minimum, maximum, average and count of  
  ↪each  
          dimension.
```

```
--boundary           Compute a hexagonal boundary that contains all points.
--dimensions arg   Use with --stats to limit the dimensions on which statistics
                   should be computed.
--dimensions "X, Y, Red"
--schema            Dump the schema of the internal point storage.
--pipeline-serialization
                   Create a JSON representation of the pipeline used to generate
                   the output.
--summary           Dump the point count, spatial reference, extrema and dimension
                   names.
--metadata          Dump the metadata associated with the input file.
```

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": 425.88999999999999
},
...
...
```

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": 637012.239999999999,
"Y": 849028.31000000006,
"Z": 431.66000000000003
},
{
"Blue": 68,
"Classification": 1,
"EdgeOfFlightLine": 0,
"GpsTime": 245381.45279923646,
"Green": 66,
"Intensity": 18,
"NumberOfReturns": 2,
"PointId": 1,
"PointSourceId": 7326,
"Red": 54,
"ReturnNumber": 1,
"ScanAngleRank": -11,
"ScanDirectionFlag": 1,
"UserData": 128,
"X": 636896.32999999996,
"Y": 849087.70000000007,
"Z": 446.3899999999999
},
...
...
```

merge

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

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

```
--files [-f] arg  Non-positional argument to specify 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.

pcl

The `pcl` command is used to invoke a PCL JSON pipeline. See *Filtering data with PCL* (page 190) for more information.

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

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

```
--input [-i] arg  Non-positional argument to specify input file ↴name.  
--output [-o] arg  Non-positional argument to specify output file ↴name.  
--pcl [-p] arg      Non-positional argument to specify pcl file name.  
--compress [-z]      Compress output data (if supported by output ↴format)  
--metadata [-m]      Forward metadata from previous stages.
```

pipeline

The `pipeline` command is used to execute *Pipeline* (page 37) JSON. See *Reading with PDAL* (page 181) or *Pipeline* (page 37) for more information.

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

--developer-debug	Enable developer debug (don't trap ↳ exceptions)
--label	A string to label the process with
--driver	Override reader driver
--input, -i	input file name
--pipeline-serialization	Output file for pipeline serialization
--validate	Validate the pipeline (including ↳ serialization), but do not write
	points
--progress	Name of file or FIFO to which stages ↳ should write progress
	information. The file/FIFO must exist. ↳ PDAL will not create the progress file.
--stdin, -s	Read pipeline from standard input

Note: The `pipeline` command can accept 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
```

random

The `random` command is used to create a random point cloud. It uses *readers.faux* (page 50) 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] arg    Non-positional argument to specify output file
           ↵name.
--compress [-z]      Compress output data (if supported by output
           ↵format)
--count arg          Number of points in created point cloud [0].
--bounds arg         Extent (in XYZ to clip output to):
--mean arg           --bounds "([xmin,xmax],[ymin,ymax],[zmin,zmax])"
--stdev arg           List of means (for --distribution normal)
           ↵normal)           --mean 0.0,0.0,0.0
           ↵normal)           --mean "0.0 0.0 0.0"
--stdev arg           List of standard deviations (for --distribution
           ↵normal)           --stdev 0.0,0.0,0.0
           ↵normal)           --stdev "0.0 0.0 0.0"
--distribution arg   Distribution type (uniform or normal) [uniform]
```

sort

The sort command uses *filters.mortonorder* (page 135) to sort data by XY values.

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

```
--input [-i] arg    Non-positional argument to specify input file
           ↵name.
--output [-o] arg   Non-positional argument to specify output file
           ↵name.
```

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] arg  Non-positional option for specifying input file_
↳ name
--output [-o] arg  Non-positional option for specifying output file/
↳ directory name
--length arg      Edge length for splitter cells.  See :ref:
↳ `filters.splitter`.
--capacity arg    Point capacity for chipper cells.  See :ref:
↳ `filters.chipper`.
```

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.

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>
```

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.

<code>--tindex</code>	Non-positional option for specifying the index file name.
<code>--filespec</code>	Non-positional option for specifying pattern of files to be indexed.
<code>--lyr_name</code>	Name of layer in which to store the the base name of the first file indexed.
<code>--tindex_name</code>	Name of the field in the feature in which indexed file name. ["location"]
<code>--ogrdriver</code>	OGR driver name. ["ESRI Shapefile"]
<code>--t_srs</code>	Spatial reference system in which to store index vector
<code>--a_srs</code>	Spatial reference assumed to be the reference for the source data. If the source data includes spatial reference
<code>--</code> <code>"EPSG:4326"</code>	information, this value is IGNORED. ["EPSG:4326"]

```
--write_absolute_path arg  Write absolute rather than relative file_
↳paths [false]
```

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  Non-positional option for specifying the index filename.
--filespec Non-positional option for specifying the merge output_
↳filename.
--polygon  Well-known text representation of geometric filter. Only
points inside the object will be written to the output_
↳file.
--bounds   Bounding box for clipping points. Only points inside_
↳the box
will be written to the output file.
--bounds "[[xmin,xmax],[ymin,ymax],[zmin,zmax]]"
--t_srs    Spatial reference system in which the output data should_
↳be
represented. ["EPSG:4326"]
```

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) (<http://www.sqlite.org>) file.

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

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 <code>type</code>
--json	JSON array of filters
--pipeline, -p	Pipeline output
--metadata, -m	Dump metadata output to the specified file
--reader, -r	Reader <code>type</code>
--writer, -w	Writer <code>type</code>

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

The `--pipeline` file name is optional. If given, the pipeline constructed from the command-line arguments will be written to disk for reuse in the [pipeline](#) (page 27).

The `--json` flag can be used to specify a JSON array of filters as if they were being specified in a [pipeline](#) (page 27). If a filename follows the flag, the file is opened and it is assumed that the file contains a valid JSON array of filter specifications. If the flag value is not a filename, the value is taken to be a literal JSON string that is the array of filters. 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 full-path 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. We can even set stage-specific parameters as shown.

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

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="[ { \\"type\\\": \\"filters.stats\\\" } ]"
```

CHAPTER FIVE

COMMUNITY

Community

PDAL's community interacts through [Mailing List](#) (page 35), [GitHub](#) (page 36), and [IRC](#) (page 36). Please feel welcome to ask questions and participate in all of the venues. The [Mailing List](#) (page 35) communication channel is for general questions, development discussion, and feedback. The [GitHub](#) (page 36) communication channel is for development activities, bug reports, and testing. The [IRC](#) (page 36) channel is for real-time chat activities such as meetings and interactive debugging sessions.

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.

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.

Subscribe

You can find the mailing list management page at

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

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.

There is also a public Gitter chat room integrated with the [GitHub](#) (page 36) repository and available at <https://gitter.im/PDAL/PDAL> or Gitter client.

See also:

[Development](#) (page 365) provides more information on how the PDAL software development activities operate.

IRC

You can find some PDAL developers on IRC on #pdal at [Freenode](#) (<http://freenode.net>). This mechanism is usually reserved for active meetings and other outreach with the community. The [Mailing List](#) (page 35) and [GitHub](#) (page 36) avenues are going to be more productive communication channels in most situations.

CHAPTER SIX

DRIVERS

Pipeline

Pipelines are the operative construct in PDAL. PDAL constructs a pipeline to perform data translation operations using [translate](#) (page 32), for example. While specific [applications](#) (page 17) are useful in many contexts, a pipeline provides some 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/>) manipulation facilities of whatever language you want.

Note: [pipeline](#) (page 27) is used to invoke pipeline operations via the command line.

Warning: As of PDAL 1.2, [JSON](#) (<http://www.json.org/>) is now the preferred specification language for PDAL pipelines. XML read support is still available at 1.2, but JSON is preferred. XML support will be dropped in a future release.

Introduction

A PDAL JSON object represents a processing pipeline.

A complete PDAL JSON data structure is always an object (in JSON terms). In PDAL JSON, an object consists of a collection of name/value pairs – also called members. For each member, the name is always a string. Member values are either a string, number, object, array or one of the literals: “true”, “false”, and “null”. An array consists of elements where each element is a value as described above.

Examples

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. 6.1: A simple pipeline to convert *LAS* (page 57) to *BPF* (page 48) while only keeping points inside the box $[0 \leq x \leq 100, 0 \leq y \leq 100]$.

A more complex PDAL pipeline, that reprojects the stage tagged A1, merges the result with B, and writes the merged output with the *writers.p2g* (page 92) plugin.:.

```
{  
  "pipeline": [  
    {  
      "type": "reproject",  
      "target": "A1",  
      "method": "identity"  
    },  
    {  
      "type": "merge",  
      "target": "B",  
      "sources": ["A1"]  
    },  
    {  
      "type": "writers.p2g",  
      "target": "output.gdb"  
    }  
  ]  
}
```

```

{
  "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.p2g",
    "filename": "output.tif"
  }
]
}

```

Definitions

- JavaScript Object Notation (JSON), and the terms object, name, value, array, and number, are defined in IETF RTC 4627, at <http://www.ietf.org/rfc/rfc4627.txt>.
- The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this documentation are to be interpreted as described in IETF RFC 2119, at <http://www.ietf.org/rfc/rfc2119.txt>.

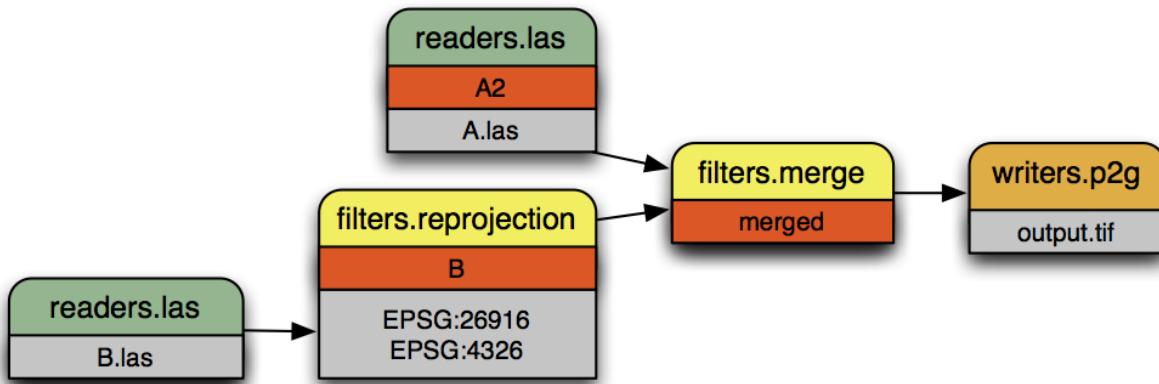


Fig. 6.2: A more complex pipeline that merges two inputs together but uses *filters.reprojection* (page 152) to transform the coordinate system of file `B.las` from `UTM` (<http://spatialreference.org/ref/epsg/nad83-utm-zone-16n/>) to `Geographic` (<http://spatialreference.org/ref/epsg/4326/>).

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 40).

Pipeline Array

- The pipeline array may have any number of string or *Stage Objects* (page 40) 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 48), *Writers* (page 74), and *Filters* (page 101).

- 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.

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).

Extended Examples

BPF to LAS

The following pipeline converts the input file from [BPF](#) (page 48) to [LAS](#) (page 82), 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.programmable](#) (page 148) 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 also:

[filters.hag](#) (page 123) describes using a specific filter to do this job in more detail.

```
{  
  "pipeline": [  
    "autzen.las",  
    {  
      "type": "ferry",  
      "dimensions": "Z=HAG"  
    },  
    {  
      "type": "programmable",  
      "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 144), and then creates the DTM using the *writers.p2g* (page 92) with only the ground returns.

```
{
  "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,
      "extract": true,
      "classify": false
    },
    {
      "type": "writers.p2g",
      "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 132) merges points from all previous readers, and the [filters.reprojection](#) (page 152) 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.

API Considerations

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

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

PDAL provides four types of stages – `pdal::Reader`, `pdal::Writer`, `pdal::Filter`, and `pdal::MultiFilter` – with the latter being hardly used (just [`filters.merge`](#) (page 132)) 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 27) 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`, `pdal::Writer`, and `pdal::Filter` are the C++ classes that define the stage types in PDAL. Readers follow the pattern of [readers.las](#) (page 57) or [readers.oci](#) (page 61), Writers follow the pattern of [writers.las](#) (page 82) or [readers.oci](#) (page 61), with Filters using [filters.reprojection](#) (page 152) or [filters.crop](#) (page 115).

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 22) for more.

Options

Options are the mechanism that PDAL uses to inform `pdal::Stage` entities how to process data. The following example sorts the data using a [Morton ordering](#) (http://en.wikipedia.org/wiki/Z-order_curve) using [filters.mortonorder](#) (page 135) and writes out a [LASzip](#) (<http://www.laszip.org>) file as the result. We use options to define the compression function for the [writers.las](#) (page 82) `pdal::Stage`.

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

Readers

Readers are data providers to *Pipeline* (page 37) operations. A reader might provide a simple file type, like *readers.text* (page 71), a complex database like *readers.oci* (page 61), or a network service like *readers.greyhound* (page 54).

Note: Readers provide *Dimensions* (page 161) to *Pipeline* (page 37). 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.

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.

Example

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

Options

filename BPF file to read [Required]

readers.buffer

The *readers.buffer* (page 49) 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 185) example, it is used to take a simple listing of points and turn them into an LAS file.

Example

See *Writing with PDAL* (page 185) for an example usage scenario for *readers.buffer* (page 49).

Options

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.

Example

```
{  
  "pipeline": [  
    {  
      "type": "readers.faux",  
      "bounds": "([0,1000000], [0,1000000], [0,100])",  
      "num_points": "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]

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

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

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

mode How to generate synthetic points. One of “constant” (repeat single value), “random” (random values within bounds), “ramp” (steadily increasing values within the bounds),

“uniform” (uniformly distributed within bounds), or “normal” (normal distribution with given mean and standard deviation). [Required]

readers.gdal

The [GDAL](http://gdal.org) (<http://gdal.org>) reader reads [GDAL readable raster](http://www.gdal.org/formats_list.html) (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 120) to copy data into known dimensions as needed.

Note: [*filters.ferry*](#) (page 120) is needed because raster data do not map to typical dimension names. For output to formats such as [*LAS*](#) (page 82), this mapping is required.

Basic Example

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

LAS Example

The following example writes a JPG as an [ASPRS LAS](#) (<http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html>)

file.

```
{  
  "pipeline": [  
    {  
      "type": "readers.gdal",  
      "filename": "./pdal/test/data/autzen/autzen.jpg"  
    },  
    {  
      "type": "filters.ferry"  
      "dimensions": "band-1=Red, band-2=Green, band-3=Blue",  
    },  
    {  
      "type": "writers.text",  
      "filename": "outputfile.txt"  
    }  
  ]  
}
```

Options

filename GDALOpen'able raster file to read [Required]

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>).

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 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]

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]

readers.greyhound

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

Example

```
{  
  "pipeline": [  
    {  
      "type": "readers.greyhound",  
      "url": "data.greyhound.io",  
      "resource": "autzen"  
    },  
    {  
      "type": "writers.text",  
      "filename": "outputfile.txt"  
    }  
  ]  
}
```

Options

url Greyhound server URL string. [Required]

resource Name of the Greyhound resource to access. [Required]

bounds Spatial bounds to query, expressed as a string, e.g. $([xmin, xmax], [ymin, ymax])$ or $([xmin, xmax], [ymin, ymax], [zmin, zmax])$. 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 54) 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 55) values.

Arbitrary logic combinations may be created using [comparison](https://docs.mongodb.com/manual/reference/operator/query-comparison/) (<https://docs.mongodb.com/manual/reference/operator/query-comparison/>) and [logical](https://docs.mongodb.com/manual/reference/operator/query-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]}}
]
```

threads Because Greyhound resources are accessed by combination of multiple HTTP requests, threaded operation can greatly increase throughput. This option sets the number of concurrent requests allowed. [Default: 4]

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. 6.3: Dimensions provided by the ILVIS2 reader

Example

```
{  
  "pipeline": [  
    {  
      "type": "readers.ilvis2",  
      "filename": "ILVIS2_GL2009_0414_R1401_042504.TXT",  
      "metadata": "ILVIS2_GL2009_0414_R1401_042504.xml"  
    },  
  ]}
```

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

Options

filename File to read from [Required]

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]

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 59) 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 82) 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.

Note: 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).

Example

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

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 build 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”]

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.

Example

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

Options

filename Filename to read from [Required]

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>) (“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 57) and *writers.las* (page 82) stages to actually read and write the data.

Example

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

Options

filename Filename to read from [Required]

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.

Example

```
{
  "pipeline": [
    {
      "type": "readers.oci",
      "query": "SELECT \r\n          1.\"OBJ_ID\", 1.\"BLK_ID\", 1.\\"BLK_EXTENT\", \r\n          1.\"PCBLK_MIN_RES\", \r\n          1.\"PCBLK_MAX_RES\", 1.\"NUM_POINTS\", \r\n          1.\"NUM_UNSORTED_POINTS\", 1.\"PT_SORT_DIM\", \r\n          1.\"POINTS\", b.cloud\r\n        FROM AUTZEN_BLOCKS 1, AUTZEN_\r\n        CLOUD b\r\n        WHERE 1.obj_id = b.id and 1.obj_id in \r\n        (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 Spatial reference system of the data being read. E.g. “EPSG:26910”.

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**]

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.

Example

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

Options

filename csd file to read [Required]

readers.pcd

The **PCD Reader** supports reading from [Point Cloud Data \(PCD\)](http://pointclouds.org/documentation/tutorials/pcd_file_format.php) (http://pointclouds.org/documentation/tutorials/pcd_file_format.php) formatted files, which are used by the [Point Cloud Library \(PCL\)](http://pointclouds.org) (<http://pointclouds.org>).

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

Example

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

Options

filename PCD file to read [Required]

readers.pgpointcloud

The **PostgreSQL Pointcloud Reader** allows you to read from a PostgreSQL database that the [PostgreSQL Pointcloud](https://github.com/pramsey/postgis) (<https://github.com/pramsey/postgis>) 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.

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

readers.ply

The **ply reader** reads 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.

The ply reader can read ASCII and binary ply files.

Example

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

Options

filename ply file to read [Required]

readers.pts

The **PTS reader** reads data from PTS files.

Example Pipeline

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

Options

filename text file to read [Required]

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.

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.

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.

Installation

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

building PDAL with optional libraries, see [the optional library documentation](#) (<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 82). 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.predicate`](#) (page 146) to filter those points earlier in the pipeline, or modify the return values with a [`filters.programmable`](#) (page 148).

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]

readers.sbet

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

Example

```
<?xml version="1.0" encoding="utf-8"?>
<Pipeline version="1.0">
  <Writer type="writers.las">
    <Option name="filename">output.las</Option>
    <Reader type="readers.sbet">
      <Option name="filename">
        sbetfile.sbet
      </Option>
    </Reader>
  </Writer>
</Pipeline>
```

Options

filename File to read from [Required]

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 98) writer. Much like the [writers.oci](#) (page 89) and [writers.pgpointcloud](#) (page 94), the SQLite driver stores data in tables that contain rows of patches. Each patch contains a number of spatially contiguous points

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
      ↪simple_blocks l, simple_cloud b\r\n
      ↪cloud = b.cloud and l.cloud in (1)\r\n
      ↪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.

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. 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.

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

Options

filename text file to read [Required]

readers.tindex

A GDAL tile index (<http://www.gdal.org/gdaltindex.html>) is an OGR (<http://gdal.org/ogr/>)-readable data source of boundary information. PDAL provides a similar concept for PDAL-readable point cloud data. You can use the *tindex* (page 29) application to generate tile index files in any format that OGR (<http://gdal.org/ogr/>) supports writing. Once you have the tile index, you can then use the *readers.tindex* (page 72) driver to automatically merge and query the data described by the tiles.

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 37) example to read and automatically merge the data.

```
{
  "pipeline": [
    {
      "type": "readers.tindex",
      "sql": "SELECT * from pdal",
      "filter_srs": "+proj=lcc +lat_1=43 +lat_2=45.5 +lat_0=41.75",
      "filter_srs": "+lon_0=-120.5 +x_0=399999.999999999 +y_0=0 +ellps=GRS80 +units=ft",
      "filter_srs": "+no_defs",
      "merge": "true",
      "filename": "index.sqlite",
      "where": "location LIKE \'%interesting.las%\'",
      "boundary": "([635629.85, 638982.55], [848999.70, 853535.43])",
      "polygon": "POLYGON ((635629.85000000 848999.70000000, 635629.85000000 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) to use to define the tile index layer.

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

boundary A 2D box to pre-filter the tile index. If it is set, it will override any **wkt** option.

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]

Writers

Writers consume data provided by [Readers](#) (page 48). 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:
<https://github.com/PDAL/PDAL/blob/master/src/Dimension.json>

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.

Example

```
{  
  "pipeline": [  
    {  
      "type": "readers.bpf"  
      "filename": "inputfile.las"  
    },  
    {  
      "type": "writers.bpf",  
      "filename": "outputfile.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 157), [filters.chipper](#) (page 105) or [filters.divider](#) (page 117). [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.

writers.derivative

The **Derivative Writer** supports writing of primary topographic attributes.

Note: This driver uses [GDAL](http://gdal.org) (<http://gdal.org>) to write the data. Only the [GeoTIFF](http://www.gdal.org/frmt_gtiff.html) (http://www.gdal.org/frmt_gtiff.html) driver is supported at this time.

Example #1

Create a single GeoTIFF with slope values calculated using the D8 method.

```
{  
  "pipeline": [  
    "inputfile.las",  
    {  
      "type": "writers.derivative",  
      "filename": "outputfile.tif",  
      "primitive_type": "slope_d8"  
    }  
  ]  
}
```

Example #2

Create multiple GeoTIFFs containing slope, hillshade, and contour curvature values.

```
{  
  "pipeline": [  
    "inputfile.las",  
    {  
      "type": "writers.derivative",  
      "filename": "outputfile_#.tiff",  
      "primitive_type": "slope_d8,hillshade,contour_curvature"  
    }  
  ]  
}
```

Options

filename GeoTiff (http://www.gdal.org/frmt_gtiff.html) file to write. [Required]

primitive_type Topographic attribute to compute. [Default: slope_d8]

- slope_d8
- slope_fd
- aspect_d8
- aspect_fd

- contour_curvature
- profile_curvature
- tangential_curvature
- total_curvature
- hillshade

edge_length Size of grid cell in X and Y dimensions using native units of the input point cloud. [Default: 15.0]

altitude Illumination altitude in degrees (hillshade only). [Default: 45.0]

azimuth Illumination azimuth in degrees (hillshade only). [Default: 315.0]

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 therefore use any [driver that supports creation of rasters](#) (http://www.gdal.org/formats_list.html).

The technique used to create the raster is a simple interpolation where each point that falls within a given radius of a raster cell center potentially contributes to the raster's value.

Note: 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` option (defaults to Z). The writer will create 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 80) option is set to a non-zero value, a square of rasters surrounding an empty cell, and the value of each non-empty surrounding is averaged using inverse distance weighting to provide a value for the subject cell. 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 the empty value of -9999.

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. [Required]

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.

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]

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>).

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 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]

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.

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 157), [*filters.chipper*](#) (page 105) or [*filters.divider*](#) (page 117). [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’, ‘LASF_Spec’, ‘liblas’, ‘laszip encoded’. 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:268910”) 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.

writers.matlab

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

The produced files have two variables, *Dimensions* and *Points*. *Dimensions* is a comma-delimited list of dimension names, and *Points* is a double array of all dimensions of every points. This output array can get very large very quickly.

Note: The Matlab writer requires the Mat-File API from MathWorks.

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]

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.

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 157), [filters.chipper](#) (page 105) or [filters.divider](#) (page 117).

clevel File complexity level (2 characters) [Default: **03**]

stype Standard type (4 characters) [Default: **BF01**]

ostaid Originating station ID (10 characters) [Default: **PDAL**]

ftitle File title (80 characters) [Default: <spaces>]

fsclas File security classification ('T', 'S', 'C', 'R' or 'U') [Default: **U**]

oname Originator name (24 characters) [Default: <spaces>]

ophone Originator phone (18 characters) [Default: <spaces>]

fsctlh File control and handling (2 characters) [Default: <spaces>]

fsclsy File classification system (2 characters) [Default: <spaces>]

idatim Image date and time (format: ‘CCYYMMDDhhmmss’). Required. [Default: AIMIDB.ACQUISITION_DATE if set or <spaces>]

iid2 Image identifier 2 (80 characters) [Default: <spaces>]

fscltx File classification text (43 characters) [Default: <spaces>]

aimidb Comma separated list of name/value pairs to complete the AIMIDB (Additional Image ID) TRE record (format name:value). Required: ACQUISITION_DATE, will default to IDATIM value. [Default: NITF defaults]

acftb Comma separated list of name/value pairs to complete the ACFTB (Aircraft Information) TRE record (format name:value). Required: SENSOR_ID, SENSOR_ID_TYPE [Default: NITF defaults]

writers.null

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

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.

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.

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_GEOMETRY 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,x> 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.

writers.p2g

The **points to grid writer** takes in a stream of point data and writes out gridded summaries of the stream. Each cell in the output grids can give one of the: minimum value, maximum value, average value, inverse distance weighted interpolation (for sparse points), or density. The points to grid writer supports creating multiple output grids simultaneously, so it is possible to generate all grid variants in one pass.

Warning: *writers.gdal* (page 78) is a replacement for *writers.p2g*. It doesn't require an externally installed library, it supports more GDAL output formats and options, and it supports the ability to write all output to a single GeoTIFF.

Note: A project called *lidar2dems* (<https://github.com/Applied-GeoSolutions/lidar2dems>) by [Applied GeoSolutions](http://www.appliedgeosolutions.com/) (<http://www.appliedgeosolutions.com/>) integrates the P2G writer and other PDAL components into a series of scripts and utilities that make it more convenient to do DEM production with PDAL.

Example

```
{  
  "pipeline": [  
    {  
      "type": "readers.las",  
      "filename": "inputfile.las"  
    },  
    {  
      "type": "writers.p2g",  
      "grid_dist_x": "6.0",  
      "grid_dist_y": "6.0",  
      "grid_dist_z": "1.0",  
      "cell_type": "average",  
      "filename": "outputfile.tif"  
    }  
  ]  
}
```

```
"grid_dist_y": "6.0",
"radius": "8.4852813742385713",
"filename": "autzen_grid",
"output_type": "min",
"output_type": "max",
"output_type": "mean",
"output_type": "idw",
"output_type": "den",
"output_format": "asc",
}
]
}
```

Options

grid_dist_x Size of grid cell in x dimension [Default: **6**]

grid_dist_y Size of grid cell in y dimension. [Default: **6**]

radius ??? [Default: **8.48528**]

filename Base file name for output files. [Required]

output_type One or many options, specifying “min”, “max”, “mean”, “idw” (inverse distance weighted), “den” (density), or “all” to get all variants with just one option. [Default: **all**]

output_format File output format to use, one of “grid”, “tif”, or “asc”. [Default: **grid**]

z Name of the ‘z’ dimension to use. [Default: ‘Z’]

bounds Custom bounds for output raster(s). If not provided, bounds will be calculated from the bounds of the input data. [Default: **none**]

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.

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

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 105).

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: **true**]

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,x> 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.

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 [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.

Use the `storage_mode` option to choose the type of ply file to write. You can choose from:

- `default`: write a binary ply file using your host's byte ordering. If you do not specify a `storage_mode`, this is the default.
- `ascii`: write an ascii file (warning: these can be HUGE).
- `little endian`: write a binary ply file with little endian byte ordering.
- `big endian`: write a binary ply file with big endian byte ordering.

Example

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

Options

`filename` ply file to write [Required]

`storage_mode` Type of ply file to write [default: host-ordered binary]

writers.rialto

The **RialtoWriter** supports writing to Rialto-formatted tiles (<http://lists.osgeo.org/pipermail/pointdown/2015-February/000001.html>).

Example

```
{  
  "pipeline": [  
    {  
      "type": "readers.las",  
      "filename": "inputfile.las"  
    },  
    {  
      "type": "filters.reprojection",  
      "out_srs": "EPSG:4326"  
    },  
    {  
      "type": "writers.rialto",  
      "max_levels": "18",  
      "overwrite": "true",  
      "filename": "outputfile.ria"  
    }  
  ]  
}
```

Options

filename The **directory** to stage the Rialto tiles in. An exception will be thrown if the directory exists, unless the overwrite option is set to true (see below). [Required]

max_levels The maximum number of levels in the quadtree. Each rectangular node at level L reduces to 4 equally-sized nodes at level L+1. Each tile at level N-1 contains 1/4 of the points contained in the level N nodes. [Default: 16]

overwrite Delete the target directory prior to writing results? [Default: false]

writers.sqlite

The **SQLite** (<http://sqlite.org>) driver outputs point cloud data into a PDAL-specific scheme that matches the approach of *readers.pgpointcloud* (page 63) and *readers.oci* (page 61).

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.

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.

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]

jscallback When producing GeoJSON, the callback allows you to wrap the data in a function, so the output can be evaluated in a <script> tag.

quote_header When producing CSV, should the column header named by quoted? [Default: true]

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: ,]

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 152) doing geographic reprojection on XYZ coordinates), while others do not interrogate the point data at all and simply reorganize or split data.

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.

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**]

filters.attribute

The attribute filter allows you to set the values of a selected dimension. Two scenarios are supported:

- Set the value of a dimension of all points to single value (use option ‘value’)
- Set points inside an OGR-readable Polygon or MultiPolygon (use option ‘datasource’)

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.attribute",
  "dimension": "Classification",
  "datasource": "attributes.shp",
  "layer": "attributes",
  "column": "CLS"
},
{
  "filename": "attributed.las",
  "scale_x": 0.0000001,
  "scale_y": 0.0000001
}
]
```

Example 2

This pipeline sets the PointSourceId of all points from ‘autzen-dd.las’ to the value ‘26’.

```
{
  "pipeline": [
    "autzen-dd.las",
    {
      "type": "filters.attribute",
      "dimension": "PointSourceId",
      "value": 26
    },
    {
      "filename": "attributed.las",
      "scale_x": 0.0000001,
      "scale_y": 0.0000001
    }
  ]
}
```

Example 3

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.attribute",
      "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. [Default: none]

value Value to apply to the dimension. [Default: none]

datasource OGR-readable datasource for Polygon or MultiPolygon data. [Default: none]

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.
[Default: none]

layer The data source's layer to use. [Defalt: first layer]

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 28) utilizes the [filters.chipper](#) (page 105) to split data by capacity.

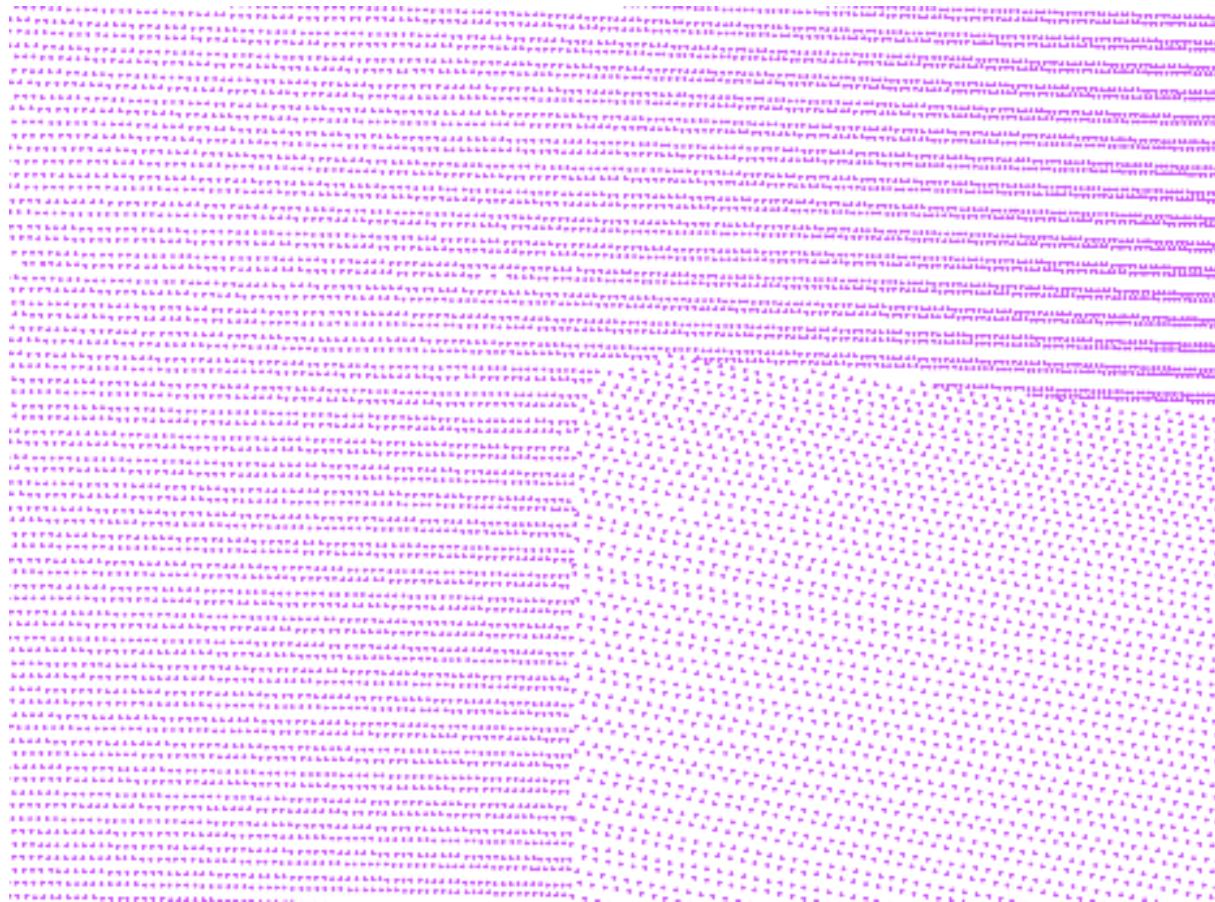


Fig. 6.4: Before chipping, the points are all in one collection.

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).

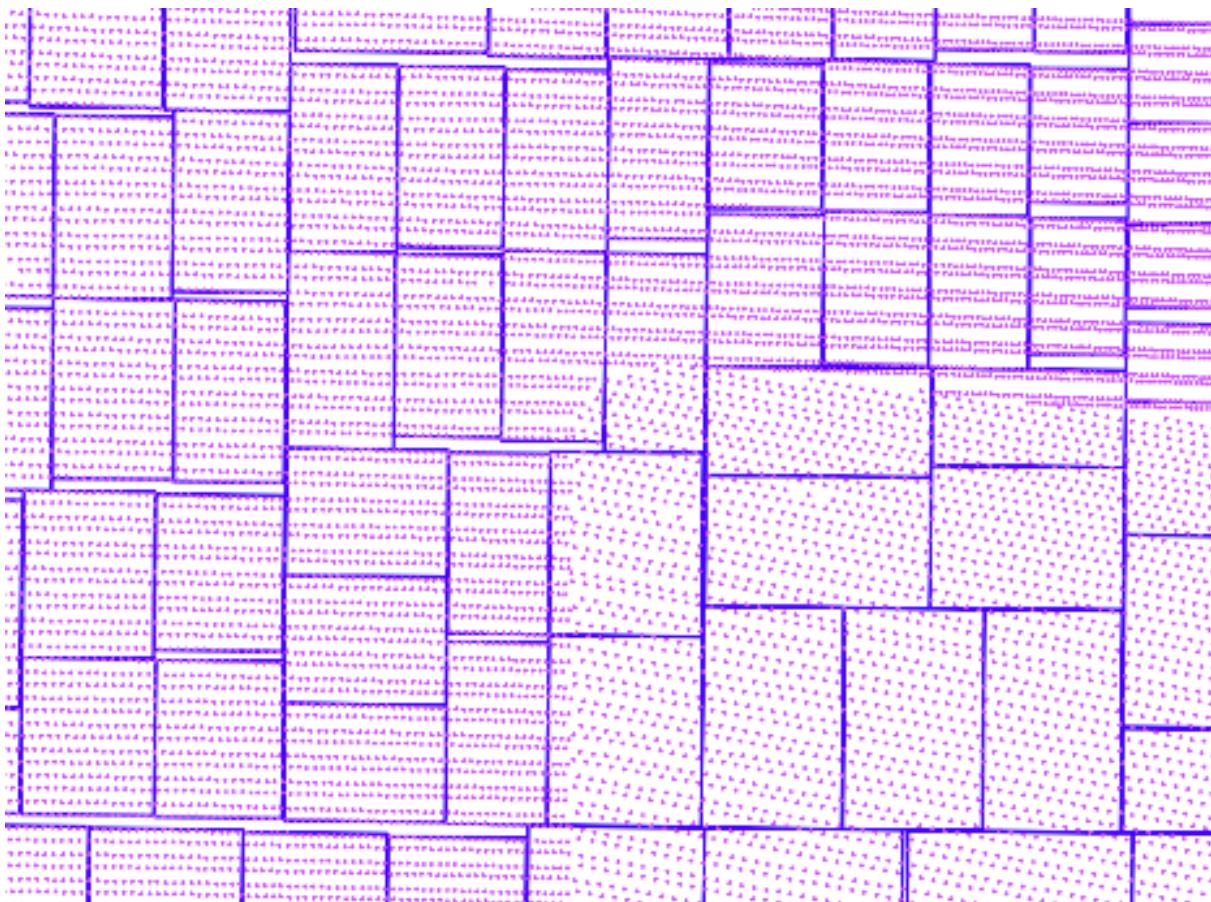


Fig. 6.5: After chipping, the points are tiled into smaller contiguous chips.

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**]

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 110) for more information.

Note: *filters.colorinterp* (page 108) will use the entire band to scale the colors.

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

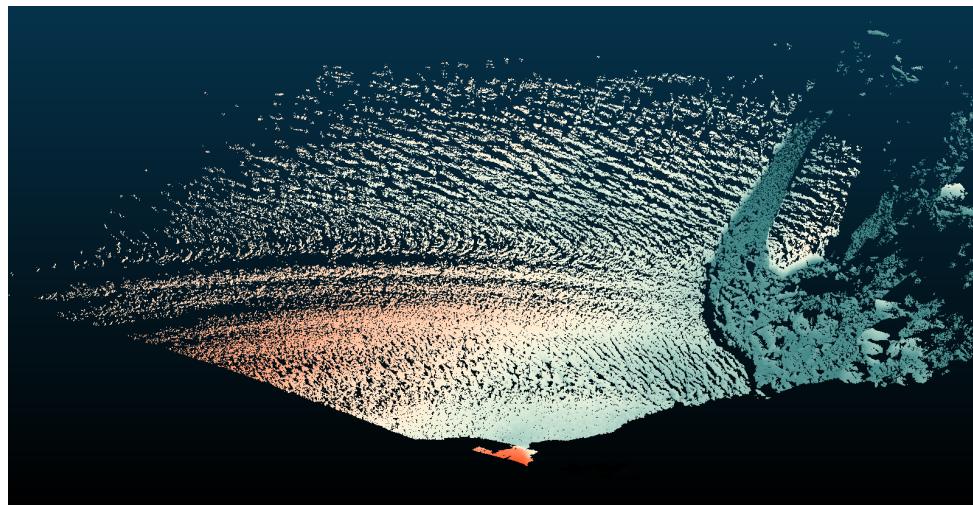


Fig. 6.6: Image data with interpolated colors based on Z dimension and pestel_shades ramp.

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 131) for discussion. [Default: false]

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

`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. 6.7: 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.

```
{  
  "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_CACHEMAX` environment variable. One driver which can have issues is when a [TIFF](http://www.gdal.org/frmt_gtiff.html) (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 111) using the striped TIFF (http://www.gdal.org/frmt_gtiff.html) with a 268mb [readers.las](#) (page 57) file will take nearly 1:54.

```
[hobu@pyro knudsen (master)]$ time ~/dev/git/pdal/bin/pdal pipeline -  
→i striped.json  
  
real 1m53.477s  
user 1m20.018s  
sys 0m33.397s
```

Setting the `GDAL_CACHEMAX` variable to a size larger than the TIFF file dramatically speeds up the color fetching:

```
[hobu@pyro knudsen (master)]$ export GDAL_CACHEMAX=500  
[hobu@pyro knudsen (master)]$ time ~/dev/git/pdal/bin/pdal pipeline -  
→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”]

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.

Example

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

filters.crop

The crop filter removes points that fall outside or inside a cropping bounding box (2D), polygon, or point+radius. 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.

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 **radius**.

radius Distance in units of common X, Y, and Z *Dimensions* (page 161) to crop circle or sphere in combination with **point**.

filters.decimation

The decimation filter retains every Nth point from an input point view.

Example

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

See also:

filters.voxelgrid (page 160) 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]

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).

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.

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.

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**]

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.

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**]

filters.ferry

The ferry filter is used to stash intermediate variables as part of processing data. For example, a common scenario is to keep both the original value and the reprojected X and Y variables in a scenario that uses the [filters.reprojection](#) (page 152) filter. In the normal case, the X and Y data would be overwritten with the new longitude and latitude values as part of the reprojection. The ferry filter will allow you to keep this around for later use.

Example

In this scenario, we are doing what is described above – stashing the pre-projection X and Y values into the *StatePlaneX* and *StatePlaneY* dimensions. Future processing, can then operate on these data.

```
{
  "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"
}
]
}

```

Options

dimensions A list of dimensions whose values should be copied to the specified dimensions.

The format of the option is <from>=<to>, <from>=<to>, ... Spaces are ignored. ‘from’ dimensions must exist and have been created by a reader or filter. ‘to’ dimensions will be created if necessary.

filters.greedyprojection

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

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.

Example

```

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

```

```
    ]  
}
```

Options

None at the moment. Relying on defaults within PCL.

filters.gridprojection

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

GridProjection is an implementation of the surface reconstruction method described in [\[Li2010\]](#) (page 431).

Example

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

Options

None at the moment. Relying on defaults within PCL.

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

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

Example

```
{
  "pipeline": [
    "input.las",
    {
      "type": "filters.hag"
    },
    {
      "type": "filters.ferry",
      "dimensions": "HeightAboveGround = Z",
    },
    {
      "type": "writers.las",
      "filename": "output.las"
    }
  ]
}
```

Options

None

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.

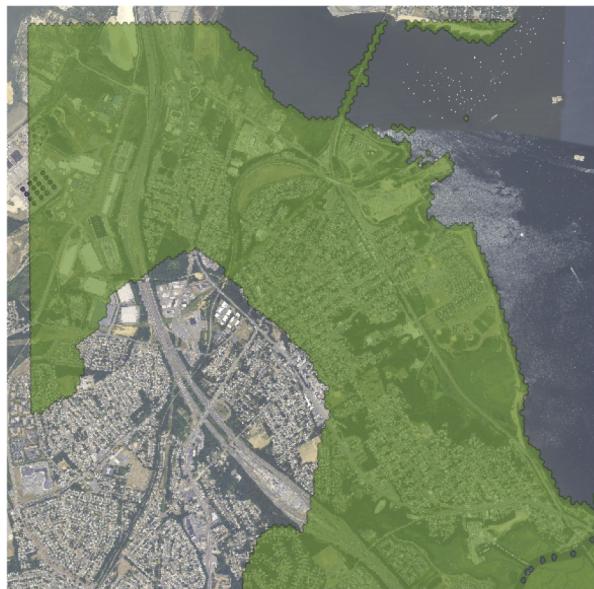


Fig. 6.8: Hexbin output shows boundary of actual points in point buffer, not just rectangular extents.

The hexbin filter reads a point stream and writes out a metadata record that contains a 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:

```
$ 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**]

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.

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.

filters.kdistance

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

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**]

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 431).

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.

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**]

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.

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.

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.

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"
  ]
}
```

filters.mongus

Filter ground returns using the approach outlined in [\[Mongus2012\]](#) (page 431).

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.

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**]

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

filters.mortonorder

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

Example

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

Notes

filters.movingleastquares

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 431). It also contains methods for upsampling the resulting cloud based on the parametric fit.

Example

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

Options

None at the moment. Relying on defaults within PCL.

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.

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**]

filters.outlier

The Outlier filter provides two outlier filtering methods: radius and statistical. These two approaches are discussed in further detail below.

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.

$$\text{outlier}_i = \begin{cases} \text{true}, & \text{if } \mu_i \geq t \\ \text{false}, & \text{otherwise} \end{cases}$$

The `classify` and `extract` options are used to control whether outlier points are labeled as noise, or removed from the output `PointView` completely.

Before outlier removal, noise points can be found both above and below the scene.

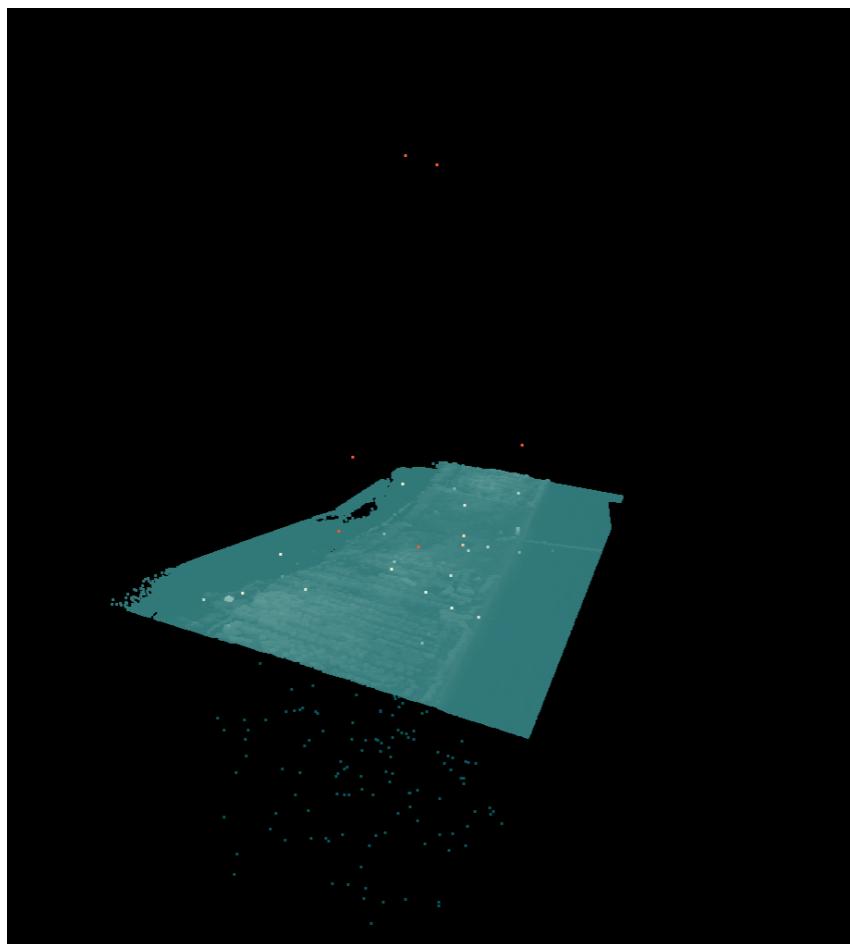
After outlier removal, the noise points are removed.

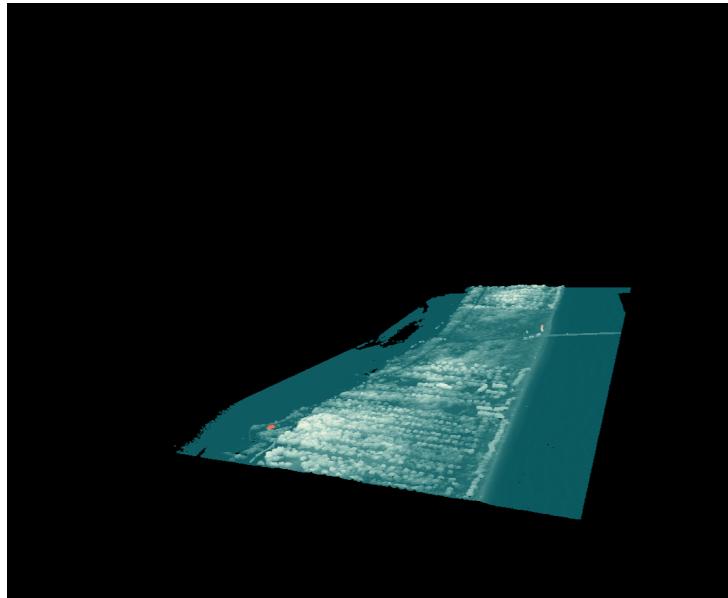
See [\[Rusu2008\]](#) (page 432) for more information.

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.

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

The `classify` and `extract` options are used to control whether outlier points are labeled as noise, or removed from the output PointView completely.

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

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**]

classify Apply classification value of 18 (LAS high noise)? [Default: **true**]

extract Extract inlier returns only? [Default: **false**]

filterspclblock

The PCL Block filter allows users to specify a block of Point Cloud Library ([PCL](http://www.pointclouds.org) (<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 190) 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.

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, `filterspclblock.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.

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 432).

Example

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

Options

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

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

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

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

cell_size Cell Size. [Default: **1**]

classify Apply classification labels? [Default: **true**]

extract Extract ground returns? [Default: **false**]

approximate Use approximate algorithm? [Default: **false**]

filters.poisson

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

Poisson is an implementation of the method described in [\[Kazhdan2006\]](#) (page 431).

Example

```
{
  "pipeline": [
    "dense.las",
    {
      "type": "filters.poisson",
      "depth": "8",
      "point_weight": "4"
    },
    {
      "type": "writers.las",
      "filename": "thinned.las",
    }
  ]
}
```

Options

depth Maximum depth of the tree used for reconstruction. [Default: **8**]

point_weight Importance of interpolation of point samples in the screened Poisson equation. [Default: **4.0**]

filters.predicate

Like the *filters.programmable* (page 148) filter, the predicate filter applies a **Python** (<http://python.org>) function to a stream of points. Points can be retained/removed from the stream by setting true/false values into a special “Mask” dimension in the output point array.

```
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
```

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.

Note: *filters.range* (page 151) 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.predicate* (page 146) and *filters.programmable* (page 148) are tools you can use for prototyping point stream processing operations.

See also:

If you want to just read a *Pipeline* (page 37) 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": [  
    "file-input.las",  
    {  
      "type": "filters.ground"  
    },  
    {  
      "type": "filters.predicate",  
      "script": "filter_pdal.py",  
      "function": "filter",  
      "module": "anything"  
    },  
    {  
      "type": "writers.las",  
      "filename": "file-filtered.las"  
    }  
  ]  
}
```

Options

script When reading a function from a separate [Python](http://python.org) (<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.

filters.programmable

The programmable filter takes a stream of points and applies a [Python](http://python.org/) (<http://python.org/>) function to each point in the stream.

The function must have two NumPy (<http://www.numpy.org/>) arrays as arguments, *ins* and *outs*. The *ins* array represents input points, the *outs* array represents output points. Each array contains all the dimensions of the point schema, for a number of points (depending on how large a point buffer the pipeline is processing at the time, a run-time consideration). Individual arrays for each dimension can be read from the input point and written to the output point.

```
import numpy as np

def multiply_z(ins,outs):
    Z = ins['Z']
    Z = Z * 10.0
    outs['Z'] = Z
    return True
```

Note that the function always returns *True*. If the function returned *False*, an error would be thrown and the translation shut down.

If you want to write a dimension that might not be available, use can use one or more *add_dimension* options.

To filter points based on a [Python](http://python.org/) (<http://python.org/>) function, use the *filters.predicate* (page 146) filter.

Example

```
{
  "pipeline": [
    "file-input.las",
    {
      "type": "filters.ground"
    },
    {
      "type": "filters.programmable",
      "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
```

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.

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.

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**]

filters.randomize

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

Example

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

filters.range

The range filter applies rudimentary filtering to the input point cloud based on a set of criteria on the given dimensions.

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. 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.

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 120) 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.

Example

```
{  
    "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"  
        }  
    ]  
}
```

Options

in_srs Spatial reference system of the input data. Express as an EPSG string (eg “EPSG:4326” for WGS86 geographic) 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 WGS86 geographic) or a well-known text string. [Required]

filters.sample

The practice of performing Poisson sampling via “Dart Throwing” was introduced in the mid-1980’s by [\[Cook1986\]](#) (page 431) and [\[Dippe1985\]](#) (page 431), and has been applied to point clouds in other software [\[Mesh2009\]](#) (page 431).

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 160)).

See also:

[filters.decimation](#) (page 116) and [filters.voxelgrid](#) (page 160) also perform decimation.

Options

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

filters.smrf

Filter ground returns using the Simple Morphological Filter (SMRF) approach outlined in [\[Pingel2013\]](#) (page 431).

Note: Our implementation of SMRF 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.smrf` differs slightly from the original paper. We weren't too happy with the performance of (our implementation of) the inpainting routine, so we started exploring some other methods.

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.

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",  
      "extract": true  
    },  
    "output.laz"  
  ]  
}
```

Options

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

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

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

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

slope Slope (rise over run). [Default: **0.15**]

threshold Elevation threshold. [Default: **0.15**]

window Max window size. [Default: **21.0**]

filters.sort

The sort filter orders a point view based on the values of a dimension. The current filter only supports sorting based on a single dimension in increasing order.

Example

```
<?xml version="1.0" encoding="utf-8"?>
<Pipeline version="1.0">
  <Writer type="writers.las">
    <Option name="filename">
      sorted.las
    </Option>
    <Filter type="filters.sort">
      <Option name="dimension">
        X
      </Option>
      <Reader type="readers.las">
        <Option name="filename">
          unsorted.las
        </Option>
      </Reader>
    </Filter>
  </Writer>
</Pipeline>
```

Options

dimension The dimension on which to sort the points.

Notes

The sorting algorithm used is not stable, meaning that one cannot chain multiple Sort filters to sort order point buffer hierarchically (say, primarily by the dimension X and secondarily by the dimension Y).

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).

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)]

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.

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.

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.

filters.voxelgrid

The Voxel Grid filter passes data through the Point Cloud Library ([PCL](#) (<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.

Example

```
{
  "pipeline": [
    "untransformed.las",
    {
      "type": "filters.voxelgrid"
    },
    {

```

```

    "type": "writers.las",
    "filename": "transformed.las"
}
]
}

```

See also:

filters.decimation (page 116) 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**]

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 101), *Writers* (page 74), and *Readers* (page 48) descriptions.

Name	Type	Description
Alpha	uint16	Alpha
Amplitude	float	This is the ratio of the received power to the power received at the detection
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 det
Deviation	float	A larger value for deviation indicates larger distortion.
EchoRange	double	Echo Range
EdgeOfFlightLine	uint8	Indicates the end of scanline before a direction change with a value of 1 - 0
ElevationCentroid	double	Elevation Centroid

Name	Type	Description
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
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 acquisition.
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
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 derived file.
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 user to control the order of points.
PointSourceId	uint16	File source ID from which the point originated. Zero indicates that the point originated from multiple sources.
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 surface.
ReflectedPulse	int32	Relative reflected pulse signal strength
ReturnNumber	uint8	Pulse return number for a given output pulse. A given output laser pulse can have multiple returns.

Name	Type	Description
Roll	float	Roll in degrees
ScanAngleRank	float	Angle degree at which the last point was output from the system, includin
ScanChannel	uint8	Scan Channel
ScanDirectionFlag	uint8	Direction at which the scanner mirror was traveling at the time of the output
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 SEVEN

TUTORIALS

Tutorials

This section provides tutorial-level information on various aspects of PDAL, from command-line use to plugin development.

Getting Started

Using PDAL with 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 185).

Note: We assume you have either *built or installed* (page 365) 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
```

PDAL Architecture Overview

Author Andrew Bell

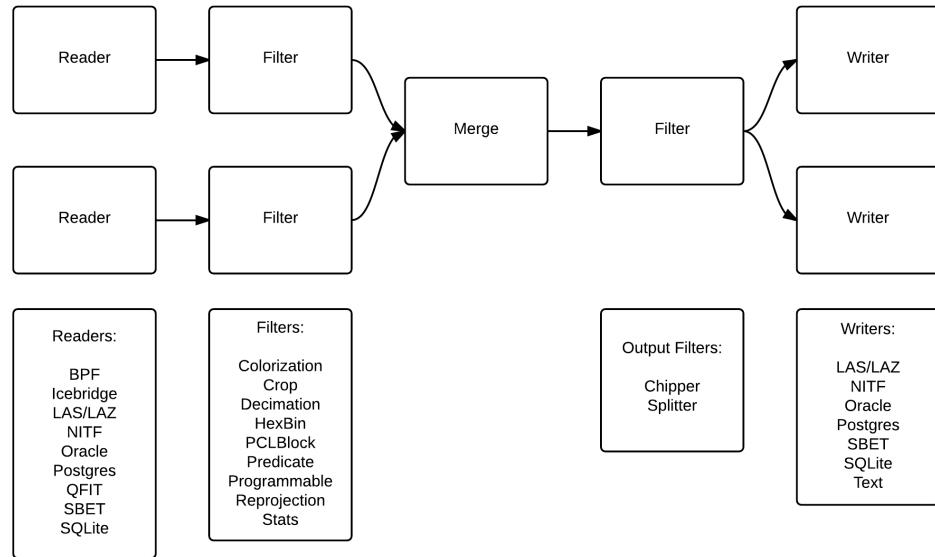
Contact andrew@hobu.co

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 XML 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 predefine 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 <<<<< Updated upstream dimension with a name and type as requested. Dimensions are described by the enumeration `pdal::Dimension::Id` and associated functions in `Dimension.hpp`. ===== dimension with a name and type as requested. Dimensions are described in a JSON file, `Dimension.json`. >>>>> Stashed changes

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 set of 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 get 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 or MultiFilter 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 processOptions(const Options& options)

PDAL allows users to specify various options at the command line and in pipeline files. Those options relevant to a stage are passed to the stage during preparation through this method. This method should extract any necessary data from the options and set data in member variables or perform other configuration as necessary. It is not recommended that options passed into this function be copied, as they may become non-copyable in a future version of the library. Handling all option processing at this point also allows an exception to be thrown in the case of an invalid option that can be properly interpreted by the pipeline.

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 pipeline.

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 of a pipeline's point layout except in 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. If a StreamPointTable is passed to execute() but a pipeline stage doesn't implement processOne(), an exception is thrown.

```
bool processOne(PointRef& ref)
```

This method allows processing of a single point. A reader will typically read a point from an input source. When a reader returns 'false' from this function, it indicates that there are no more points to be read. When a filter returns 'false' from this function, it indicates that the point just processed should be filtered out and not passed to subsequent stages for processing.

Implementing a Reader

A reader is a stage that takes input from a point cloud format supported by PDAL and loads points into a point table through a point view.

A reader needs to register or assign those dimensions that it will reference when adding point data to the point table. Dimensions that are predefined in PDAL can be registered by using the point table's registerDim() method. Dimensions that are not predefined can be added using assignDim(). If dimensions are determined as named entities from a point cloud source, it may not be known whether the dimensions are predefined or not. In this case the function

registerOrAssignDim() can be used. When a dimension is assigned, rather than registered, the reader needs to inform PDAL of the type of the variable using the enumeration Dimension::Type.

In this example, the reader informs the point table's layout that it will reference the dimensions X, Y and Z.

```
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 filehandles 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 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. Some current writers do not produce correct output when provided with multiple point views. Users should use a merge filter immediately prior to such writers to avoid errors. 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);
}
```

Using PDAL

Reading with PDAL

Author Bradley Chambers

Contact brad.chambers@gmail.com

Date 01/21/2015

Contents

- *Reading with PDAL* (page 181)
 - *Introduction* (page 181)
 - *A basic inquiry example* (page 182)
 - *A conversion example* (page 183)
 - * *Metadata (page 403)* (page 184)
 - *A pipeline (page 27) example* (page 184)
 - * *Simple conversion* (page 185)
 - * *Loop a directory and filter it through a pipeline* (page 185)

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 421) 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) (<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) (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)

(<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 403), 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` 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) [UUID](#) (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 403) that is carried per-stage throughout the PDAL [processing pipeline](#) (page 37). We can capture this metadata using the [info](#) (page 22) utility.

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

This produces metadata that looks like [this](#) (page 407). 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 403) contains much more detail of metadata workflow in PDAL.

A pipeline example

The full power of PDAL comes in the form of [pipeline](#) (page 27) invocations. While [translate](#) (page 32) 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 37), but the [Pipeline](#) (page 37) document contains more detailed exposition of the topic.

Note: The [pipeline](#) (page 27) document contains detailed examples and background

information.

The [pipeline](#) (page 27) PDAL utility is one that takes in a .json file containing [pipeline](#) (page 27) description that defines a PDAL processing pipeline. Options can be given at each pdal::Stage 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/>) document defines a [Pipeline](#) (page 37) that takes the `file.las` [ASPRS LAS](#) (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
```

Writing with PDAL

Author Bradley Chambers

Contact brad.chambers@gmail.com

Date 01/21/2015

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 48) that conforms to PDAL's `pdal::Stage`, you can implement a simple read-filter-write pipeline using *Pipeline* (page 37) 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);

    BufferReader reader;
    reader.addView(view);

    StageFactory factory;

    // Set second argument to 'true' to let factory take ownership of
    // stage and facilitate clean up.
    Stage *writer = factory.createStage("writers.las");

    writer->setInput(reader);
    writer->setOptions(options);
    writer->prepare(table);
    writer->execute(table);
}
```

Take a closer look. We will need to include several PDAL headers.

```
#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);

    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);
}
```

Compiling and running the program

Note: Refer to *Compilation* (page 365) 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 2.8.12)
project(WritingTutorial)

find_package(PDAL 1.0.0 REQUIRED CONFIG)

set(CMAKE_CXX_FLAGS "-std=c++11")
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 *Using PDAL with CMake* (page 165) 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
```

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 141) for details on PDAL's PCL Block filter.

Contents

- [Filtering data with PCL](#) (page 190)
 - [Introduction](#) (page 191)
 - [Quick Start](#) (page 191)
 - [PDAL Pipeline kernel](#) (page 192)
 - [PDAL PCL kernel](#) (page 193)
 - [Examples](#) (page 194)
 - * [Simple point cloud cropping](#) (page 194)
 - * [Point cloud cropping with outlier removal](#) (page 195)
 - * [Ground return filtering](#) (page 196)

Quick Start

The [Quickstart](#) (page 9) 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 37), [pipeline](#) (page 27), and [Reading with PDAL](#) (page 181) documents contain detailed examples and background information.

The [filters.pclblock](#) (page 141) 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>) 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 197) for a more detailed explanation of the PMF parameters.

Identifying ground returns using ProgressiveMorphologicalFilter 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 144), which is a native PDAL filter. [`ground`](#) (page 20) has been retained, but now calls [`filters.pmf`](#) (page 144) 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.

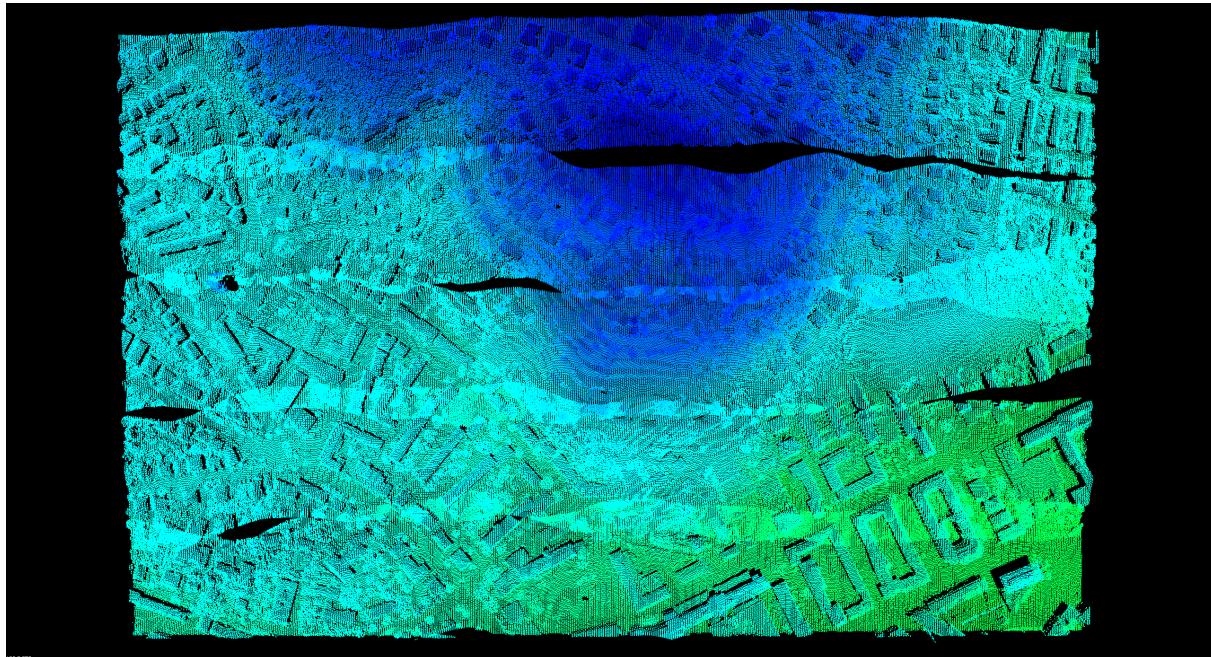
Background

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>) 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) tutorial on the PCL website.

We have chosen to demonstrate the algorithm using data from the 2003 report “[ISPRS Comparison of Filters](#).“ 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>) 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) and save it somewhere to disk.



Using the Ground kernel

The *pdal ground* (page 20) 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 CSite1_orig-utm.laz -o CSite1_orig-utm-ground.laz --  
  visualize
```

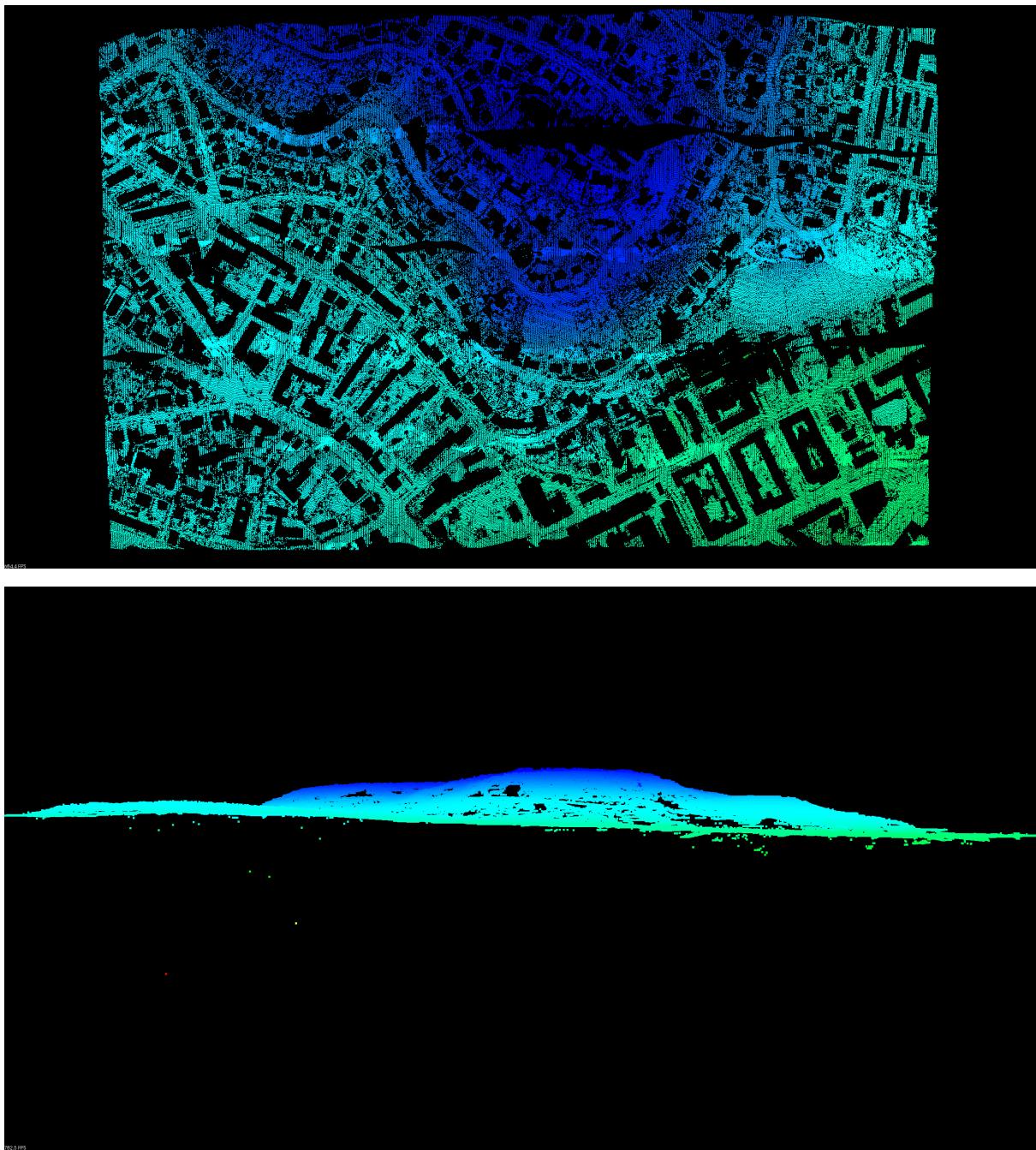
Note: In this tutorial, we use `--visualize` to visualize results, but this is only available if PCL is built with VTK and visualization support. If your install does not support VTK/visualization, simply drop `--visualize` and visualize the result with the viewer of your choice.

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 --  
→visualize -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

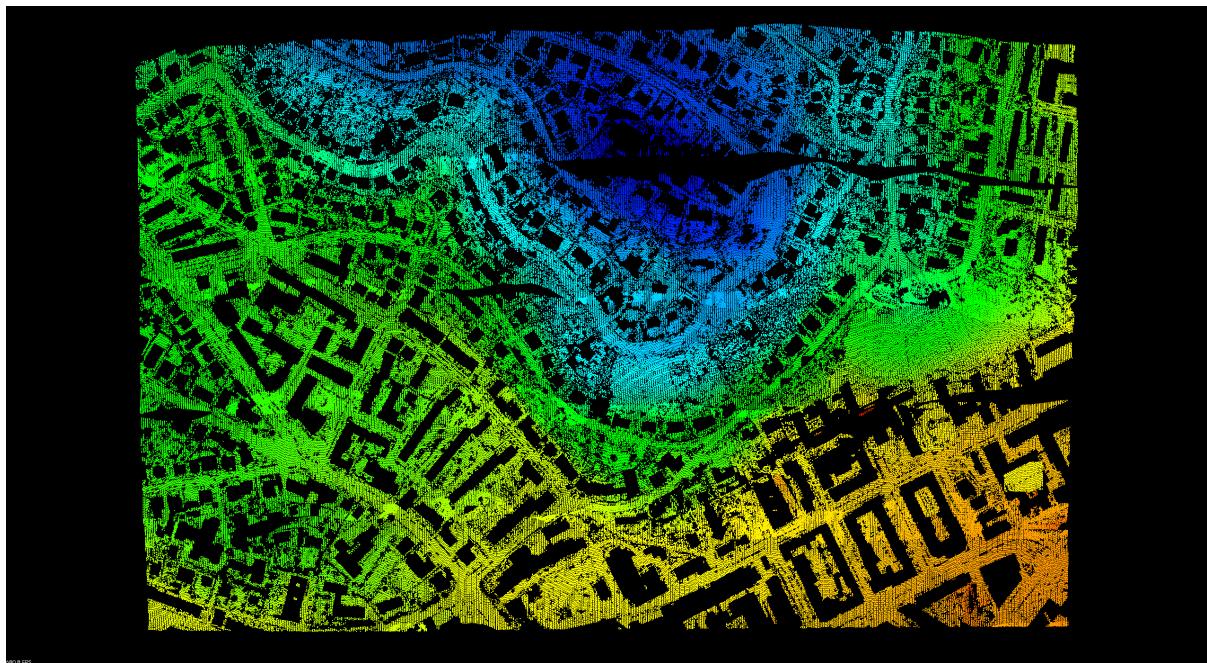
preprocess with an outlier removal step. The command is nearly identical, replacing ground with pcl and adding a pipeline JSON specified with -p.

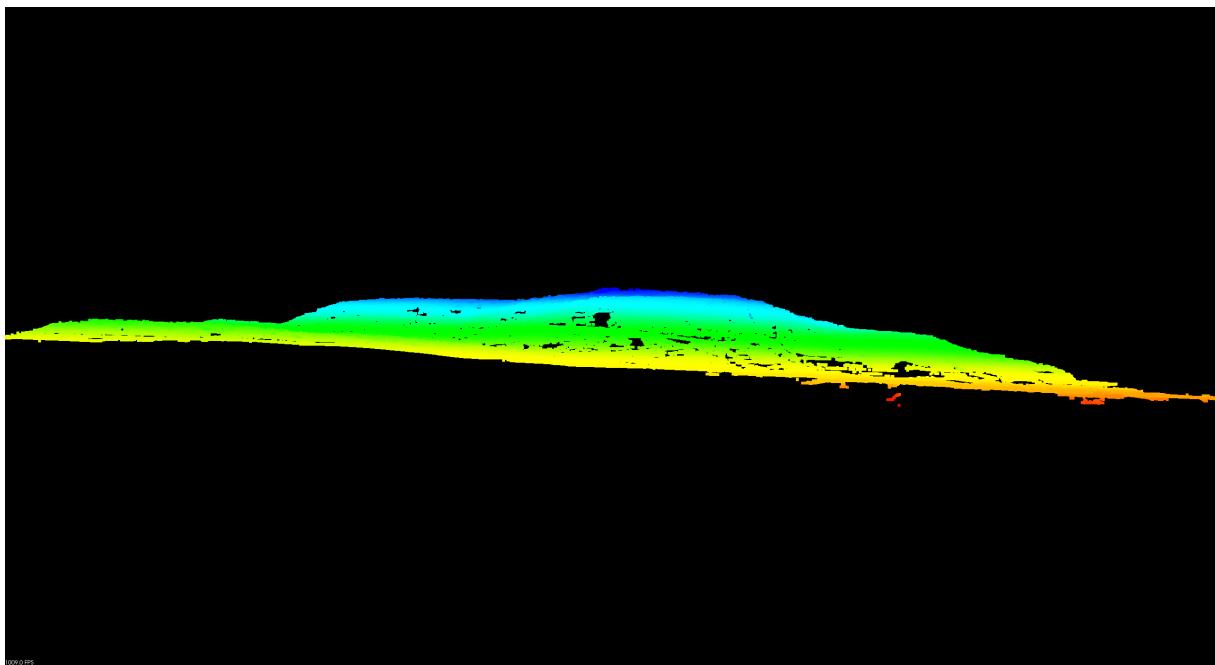
```
{  
    "pipeline": {  
        "name": "Progressive Morphological Filter with Outlier Removal",  
        "version": 1.0,  
        "filters": [{  
            "name": "StatisticalOutlierRemoval",  
            "setMeanK": 8,  
            "setStddevMulThresh": 3.0  
        }, {  
            "name":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

```
$ pdal pcl -i CSitel_orig-utm.laz -o CSitel_orig-utm-ground.laz -p  
→sor-pmf.json --visualize -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 CSitel_orig-utm.laz -o CSitel_orig-utm-ground.laz -p
  ↳sor-pmf2.json --visualize -v4

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

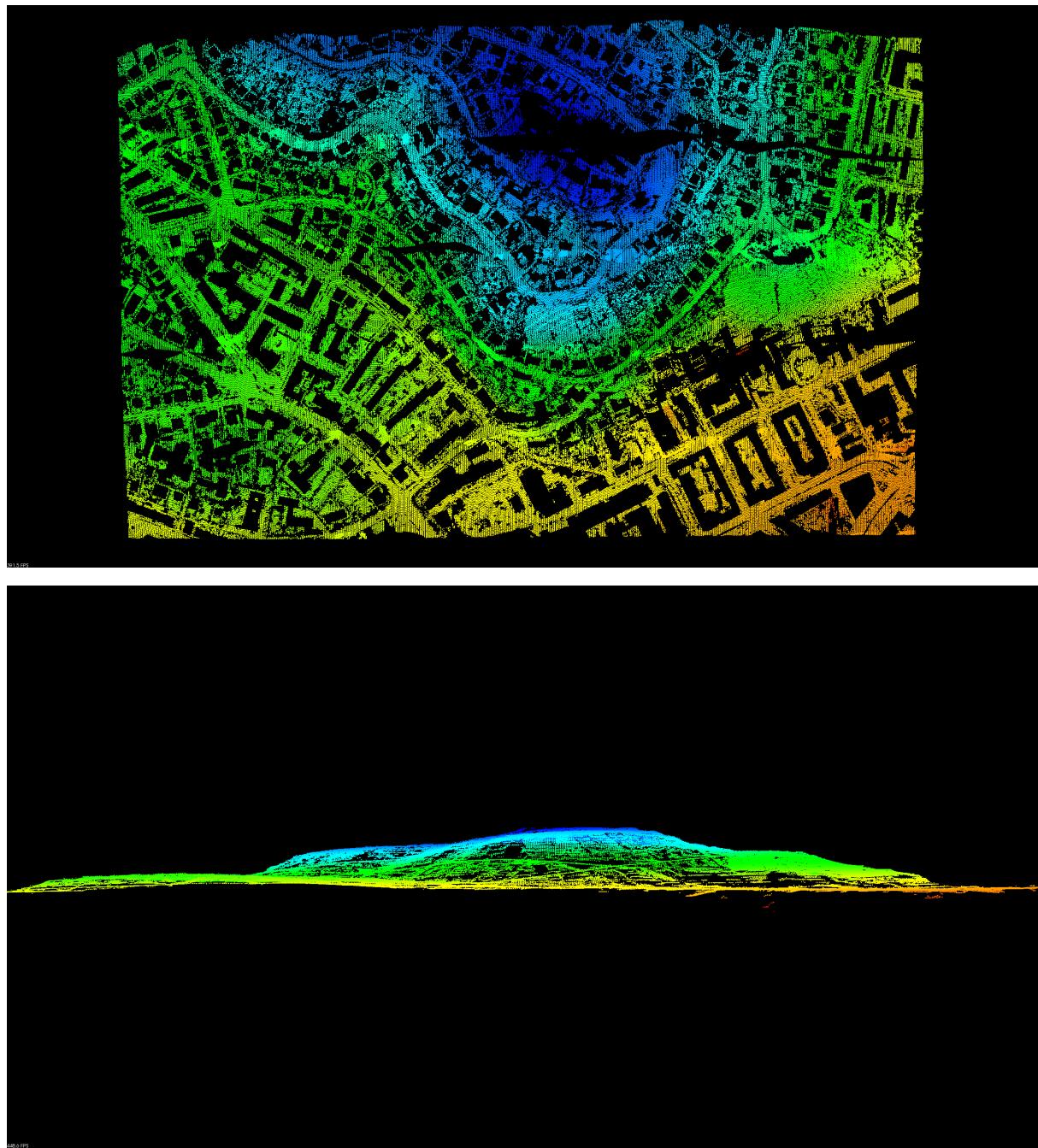
Step 1) StatisticalOutlierRemoval

  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.



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 206)
 - [*Introduction*](#) (page 206)
 - [*Example Data*](#) (page 206)
 - [*Stage Operations*](#) (page 207)
 - [*Data Preparation*](#) (page 207)
 - [*Pipeline*](#) (page 208)
 - [*Processing*](#) (page 210)
 - [*Conclusion*](#) (page 210)

Example Data

This tutorial utilizes the Autzen dataset. In addition to typical PDAL software (fetch it from [*Download*](#) (page 5)), 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.attribute* (page 103) 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 151), which allows you to keep or reject points from the set that match given criteria.

See also:

filters.predicate (page 146) allows you to construct sophisticated logic for keeping or rejecting points in a more expressive environment (**Python** (<http://www.python.org>)).

Data Preparation



Fig. 7.1: Autzen Stadium, a 100 million+ point cloud file.

The data are mixed in two different coordinate systems. The **LAZ** (page 57) 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. 7.2: We want to clip out the polygon in the upper right hand corner. We can view the [GeoJSON](http://geojson.org) (<http://geojson.org>) geometry using something like [QGIS](http://qgis.org) (<http://qgis.org>)

Pipeline

A PDAL [*Pipeline*](#) (page 37) is how you define a set of actions to happen to data as they are read, filtered, and written.

```
{
  "pipeline": [
    "autzen.laz",
    {
      "type": "filters.attribute",
      "dimension": "Classification",
      "datasource": "attributes.vrt",
      "layer": "OGRGeoJSON",
      "column": "CLS"
    },
    {
      "type": "filters.range",
      "limits": "Classification[5:5]"
    },
    "output.las"
  ]
}
```

- *readers.las* (page 57): 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.attribute* (page 103): Using the VRT we defined in *Data Preparation* (page 207), read attribute polygons out of the data source and assign the values from the `CLS` column to the `Classification` field.
- *filters.range* (page 151): 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 82): 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.attribute* (page 103). Any valid dimension name could work, but most LiDAR softwares will display categorical coloring for the `Classification` field, and we can leverage that

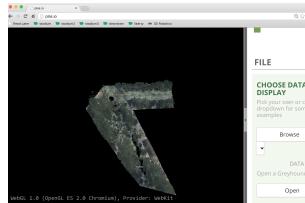
behavior in this scenario.

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 37).

```
$ 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.attribute` (page 103) and `filters.range` (page 151) stages to clip points with shapefiles.

Calculating Normalized Heights

Author Bradley Chambers

Contact brad.chambers@gmail.com

Date 11/11/2015

This tutorial will describe the creation of a new filter for calculating normalized heights, `filters.height`.

Note: `filters.height` required PCL and has since been replaced by [`filters.hag`](#) (page 123), 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

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 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. While this may not be the most accurate approach, it is available in PDAL today, and we hope it will inspire you to implement your own methods!

Approach

For the height filter, we only assume that our input point cloud has an already existing Classification dimension with some subset of points marked as ground (Classification=2). This could, for example, be generated by [`filters.pmf`](#) (page 144) (see [*Identifying ground returns using ProgressiveMorphologicalFilter segmentation*](#) (page 197)), but you can use whichever method you choose, as long as the ground returns are marked.

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).

For the full source, please see `HeightFilter.cpp` in the `plugins/pcl/filters` folder. Below, we dissect the key aspects of the algorithm and its implementation.

The bulk of our processing is actually taking place within PCL. For convenience, we've defined:

```
typedef pcl::PointCloud<pcl::PointXYZ> Cloud;
```

Our first step is to convert the filter's incoming PDAL `PointView` to a PCL `PointCloud`, which requires that we first calculate our bounds so that we can subtract our XYZ offsets in the conversion step.

```
BOX3D bounds;
view.calculateBounds(bounds);

Cloud::Ptr cloud_in(new Cloud);
pclsupport::PDALtoPCD(std::make_shared<PointView>(view), *cloud_in, ↴
    ↴bounds);
```

Next, we will create two vectors of indices - one for ground returns, one for non-ground returns - and make our first pass through the point cloud to populate these.

```
pcl::PointIndices::Ptr ground(new pcl::PointIndices());
ground->indices.reserve(view.size());

std::vector<PointId> nonground;
nonground.reserve(view.size());

for (PointId id = 0; id < view.size(); ++id)
{
    double c = view.getFieldAs<double>(Dimension::Id::Classification, id);

    if (c == 2)
        ground->indices.push_back(id);
    else
        nonground.push_back(id);
}
```

With our ground indices identified, we can use PCL to extract the ground returns into a new `PointCloud`.

```
pcl::ExtractIndices<pcl::PointXYZ> extract;
extract.setInputCloud(cloud_in);
extract.setIndices(ground);

Cloud::Ptr cloud_ground(new Cloud);
extract.setNegative(false);
extract.filter(*cloud_ground);
```

We repeat the extraction now, flipping `setNegative` from false to true to extract the non-ground returns into a new `PointCloud`.

```
Cloud::Ptr cloud_nonground(new Cloud);
extract.setNegative(true);
extract.filter(*cloud_nonground);
```

To compute the normalized height, we wish to find the nearest ground point for each non-ground point. Here, we achieve this by using a nearest neighbor interpolation scheme. One may prefer to use a more sophisticated interpolation scheme, but that is beyond the scope of this tutorial. We begin by defining model coefficients that will allow us to project the ground and non-ground clouds into the XY plane.

```
pcl::ModelCoefficients::Ptr coefficients(new pcl::
    ModelCoefficients());
coefficients->values.resize(4);
coefficients->values[0] = coefficients->values[1] = 0;
coefficients->values[2] = 1.0;
coefficients->values[3] = 0;
```

We can now project the ground points

```
pcl::ProjectInliers<pcl::PointXYZ> proj;
proj.setModelType(pcl::SACMODEL_PLANE);

Cloud::Ptr cloud_ground_projected(new Cloud);
proj.setInputCloud(cloud_ground);
proj.setModelCoefficients(coefficients);
proj.filter(*cloud_ground_projected);
```

followed by the non-ground points

```
Cloud::Ptr cloud_nonground_projected(new Cloud);
proj.setInputCloud(cloud_nonground);
proj.setModelCoefficients(coefficients);
proj.filter(*cloud_nonground_projected);
```

Next, we create a KdTree to accelerate our nearest neighbor search. The tree is composed of only ground returns, as our non-ground returns will serve as query points for the nearest neighbor search.

```
pcl::search::KdTree<pcl::PointXYZ>::Ptr ground_tree;
ground_tree.reset(new pcl::search::KdTree<pcl::PointXYZ> (false));
ground_tree->setInputCloud(cloud_ground_projected);
```

We iterate over each of our projected non-ground points, searching for our nearest neighbor in the ground points. Using the indices of each the query (non-ground) and nearest neighbor (ground), we can retrieve the Z dimension from the input cloud, compute the height, and set this field in our original PointView.

```
for (int i = 0; i < cloud_nonground_projected->size(); ++i)
{
    pcl::PointXYZ nonground_query = cloud_nonground_projected->
    ~points[i];
    std::vector<int> neighbors(1);
    std::vector<float> sqr_distances(1);
    ground_tree->nearestKSearch(nonground_query, 1, neighbors, sqr_
    ~distances);

    double nonground_Z = view.getFieldAs<double>(Dimension::Id::Z, ~
    nonground[i]);
    double ground_Z = view.getFieldAs<double>(Dimension::Id::Z, ~
    ground->indices[neighbors[0]]);
    double height = nonground_Z - ground_Z;

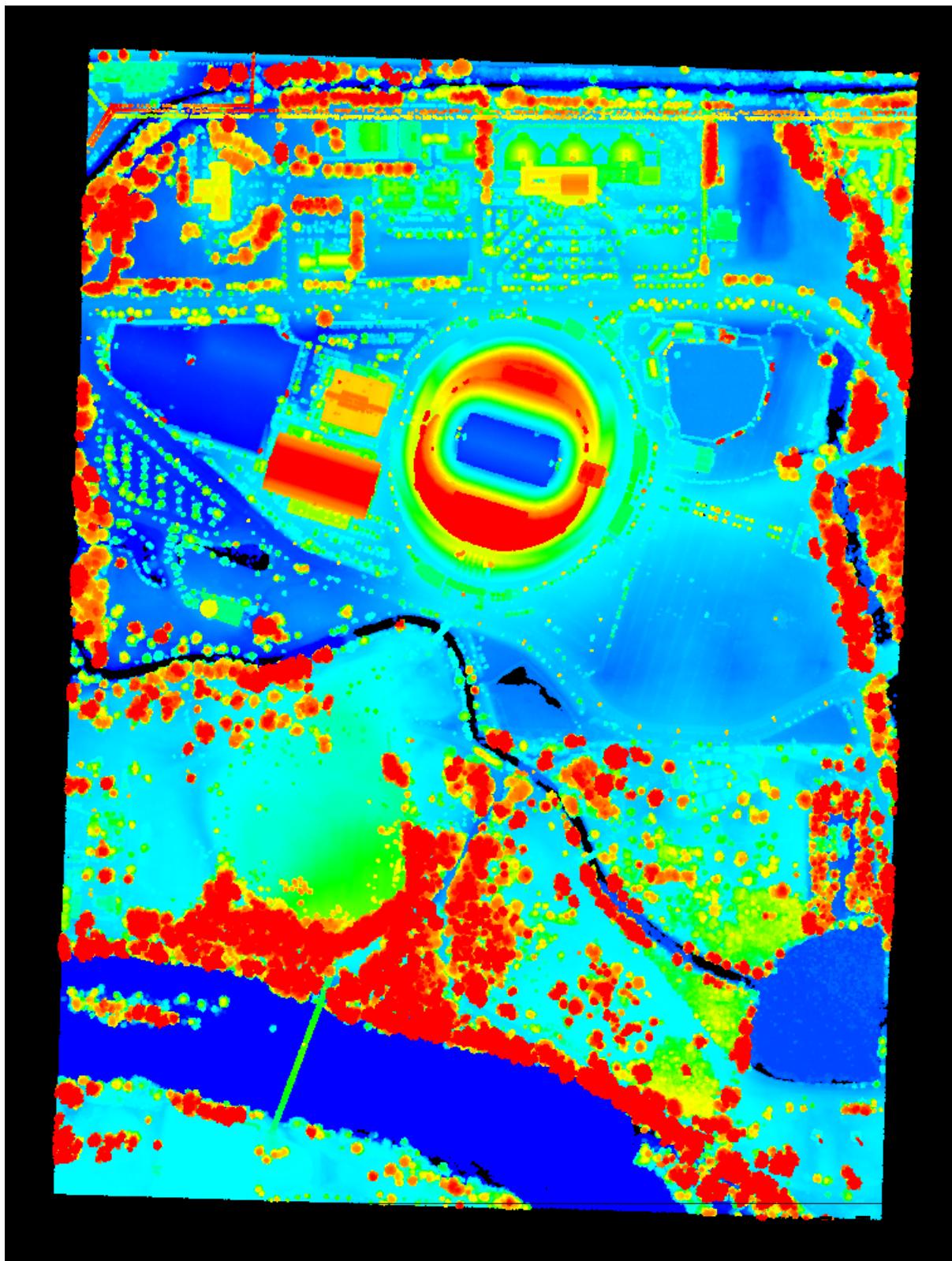
    view.setField(m_heightDim, nonground[i], height);
}
```

The only thing left is to set the height field to 0.0 for each of the ground points.

```
for (auto const& ground_idx : ground->indices)
    view.setField(m_heightDim, ground_idx, 0.0);
```

Example #1

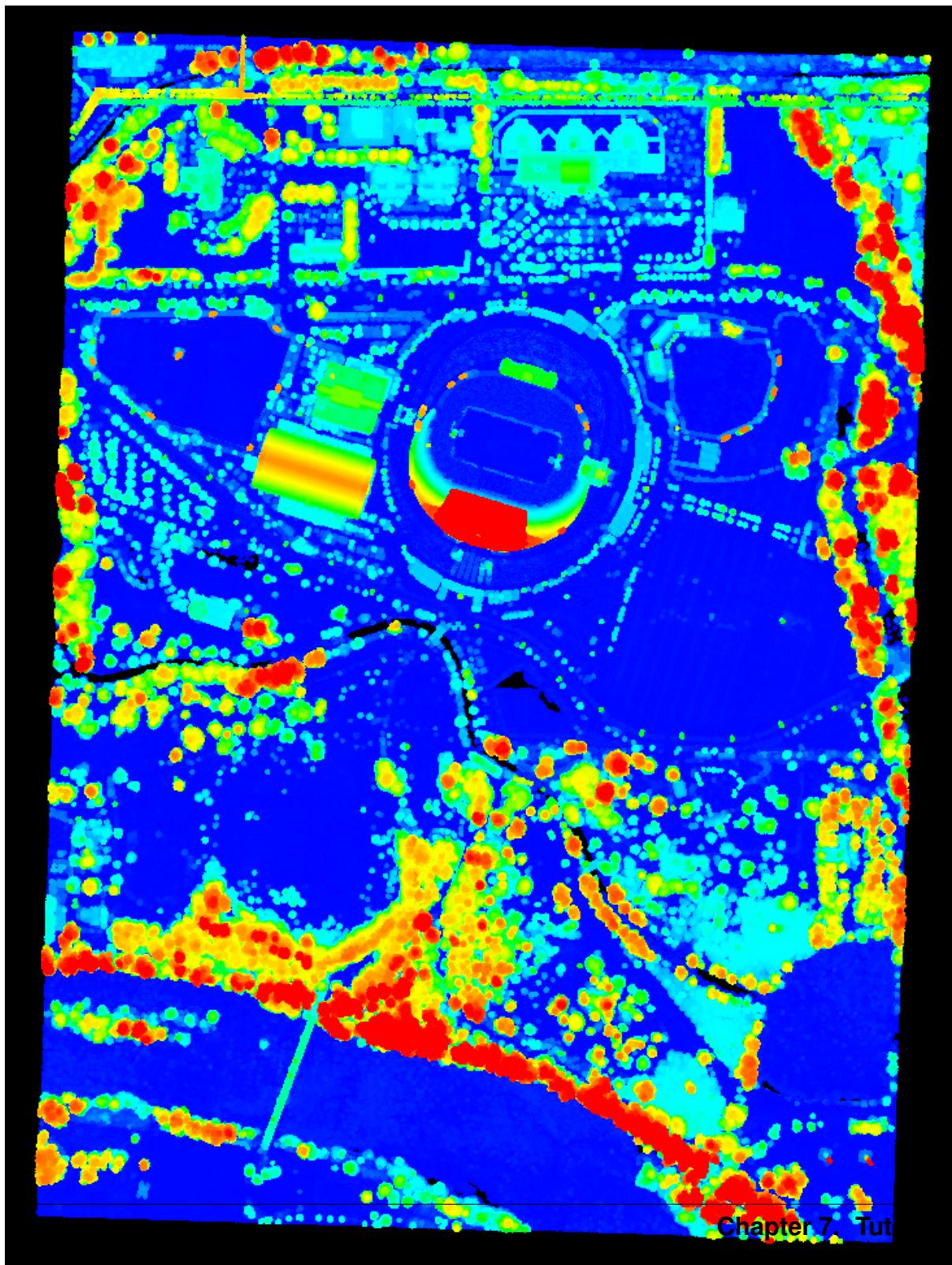
Using the autzen dataset (here shown colored by elevation)



we run the following PDAL CLI command

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

The result, when colored by the normalized height instead of elevation is



Example #2

If you'd like to overwrite your Z values, follow the height filter with [*filters.ferry*](#) (page 120).

```
$ pdal translate input.laz output-height-as-Z.bpf height ferry \
--writers.bpf.output_dims="X,Y,Z" --filters.ferry.dimensions=
↪"HeightAboveGround=Z"
```

Example #3

If you don't yet have points classified as ground, start with [*filters.pmf*](#) (page 144).

```
$ pdal translate input.laz output-ground-height.bpf ground height \
--writers.bpf.output_dims="X,Y,Z,HeightAboveGround"
```

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 154).

Note: *filters.dartsample* required PCL and has since been replaced by [*filters.sample*](#) (page 154), 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.

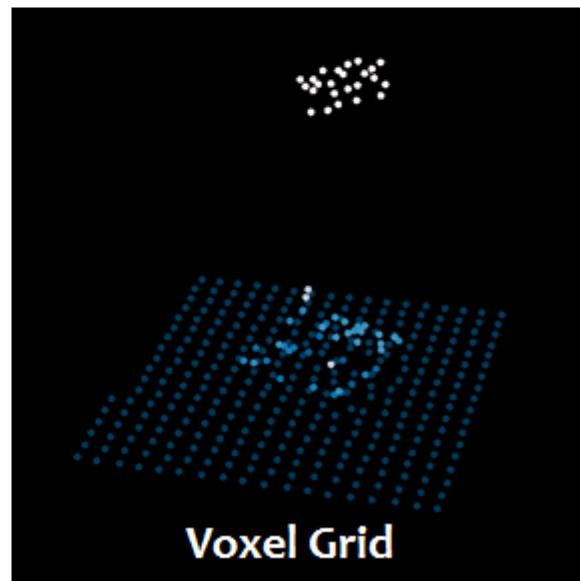
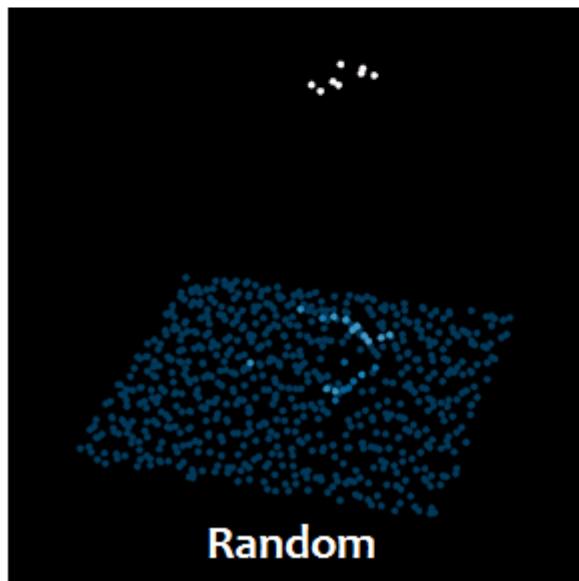
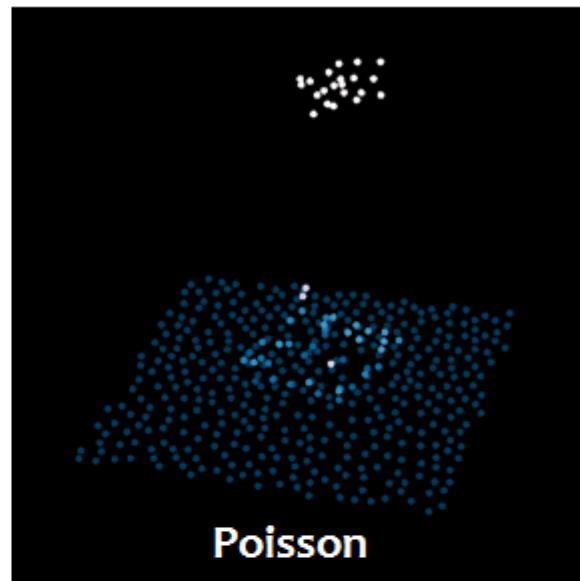
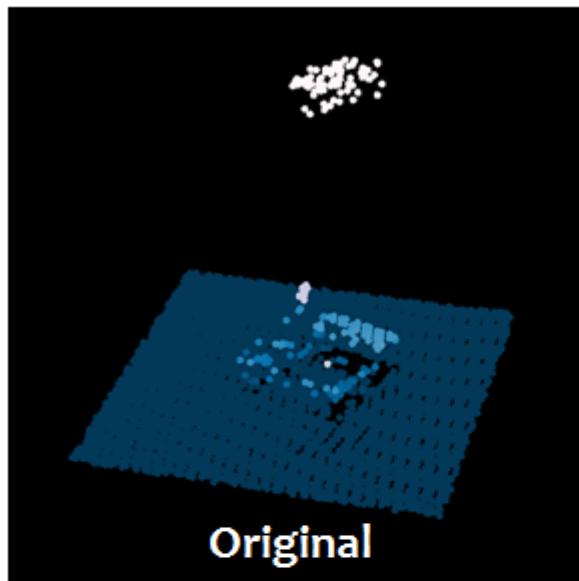
```
$ pdal translate autzen.laz autzen-dart-sampled.laz dartsample \
--filters.dartsample.radius=3
```

For comparison, we will process autzen using both `filters.decimation` (page 116), with a step size of 6, and `filters.voxelgrid` (page 160), 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
```

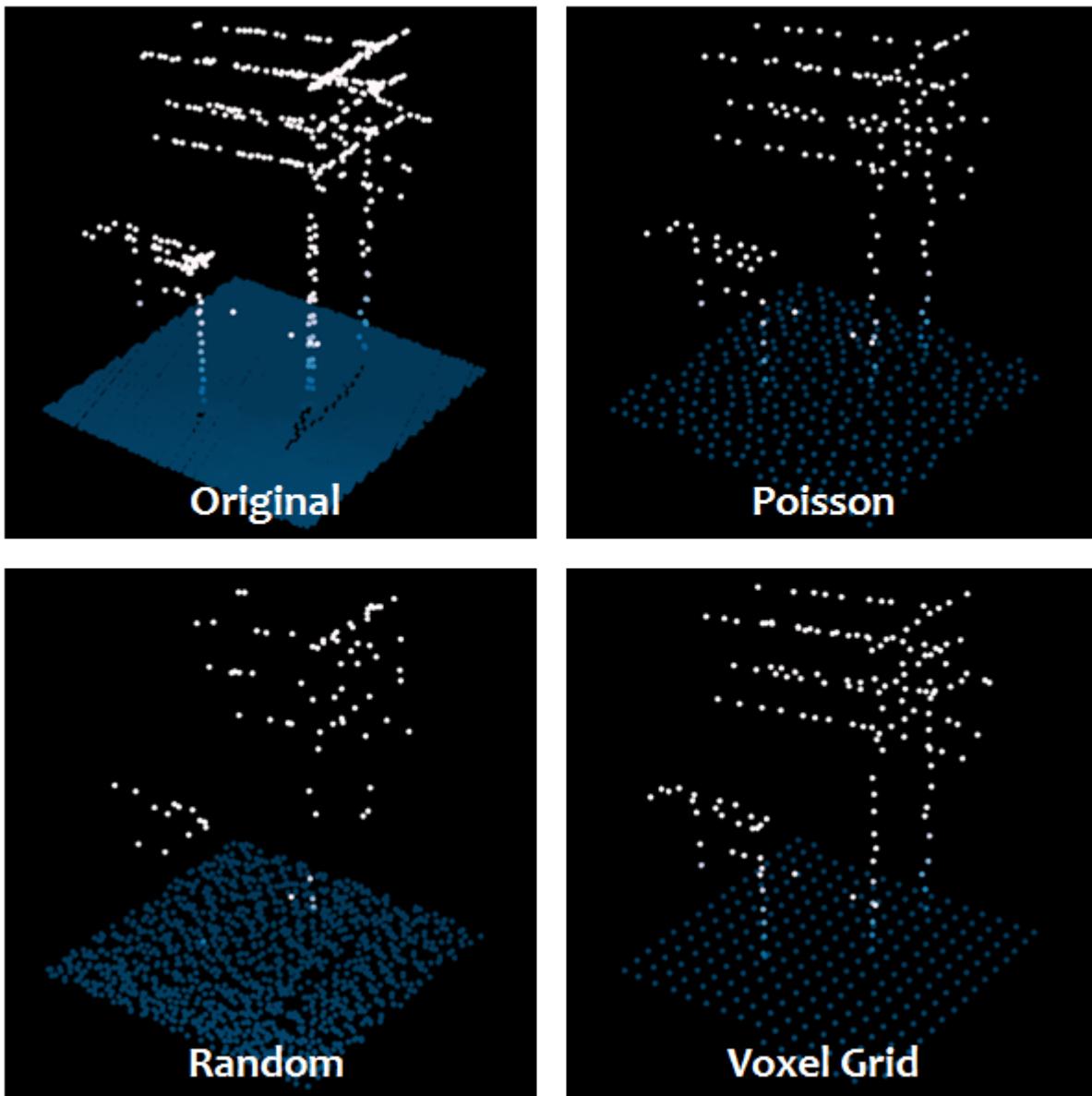
```
$ pdal translate autzen.laz autzen-voxelgrid-sampled.laz voxelgrid \
--filters.voxelgrid.leaf_x=4.5 --filters.voxelgrid.leaf_y=4.5 --
˓→filters.voxelgrid.leaf_z=4.5
```

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.



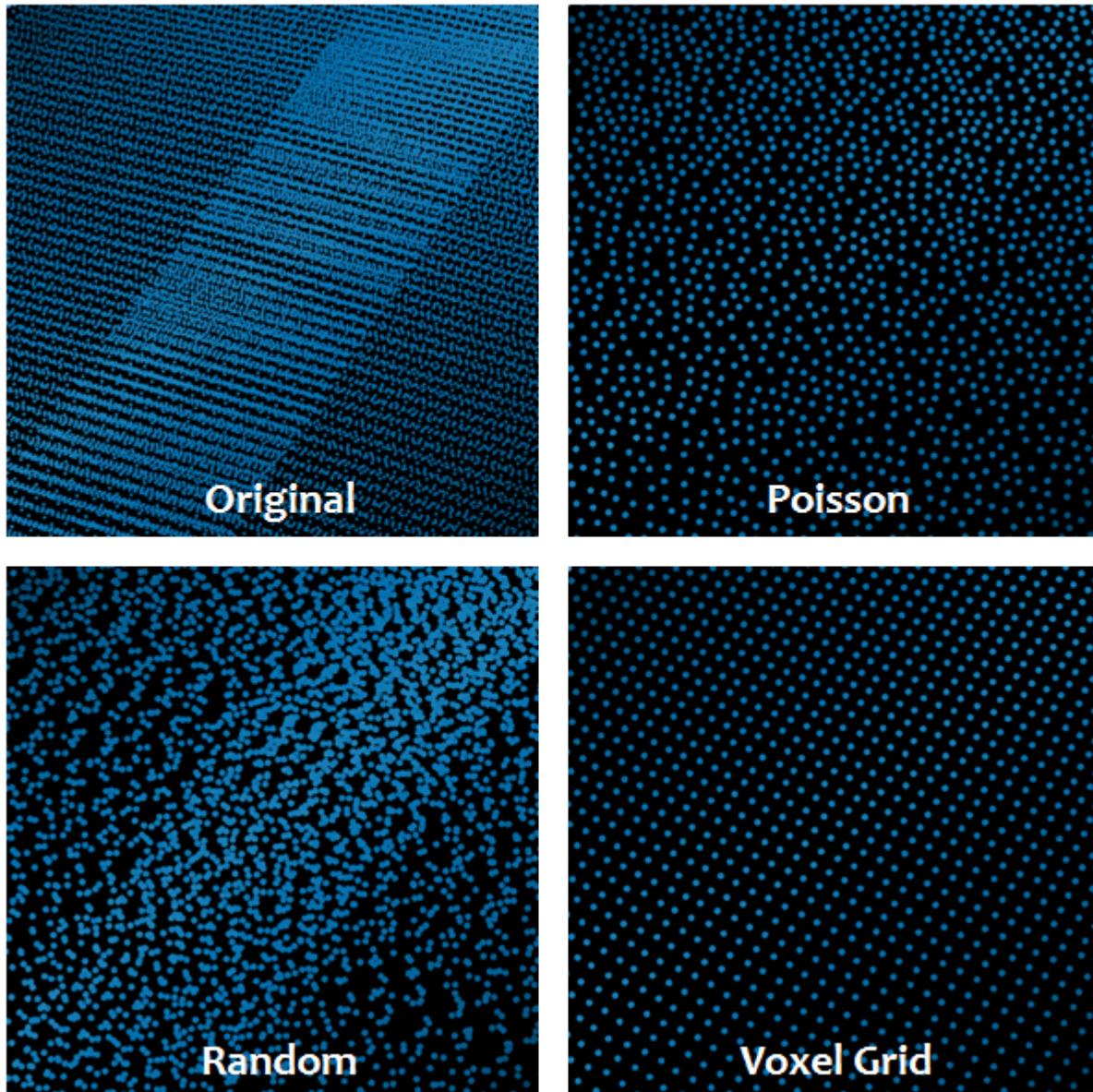
The Poisson and voxel gridded clouds both appear to do a good job retaining key elements of the structure, while significantly reducing the number of points allocated to the ground plane. The randomly sampled cloud however, exhibits the opposite behavior, noticeably degrading the structure, while heavily 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.



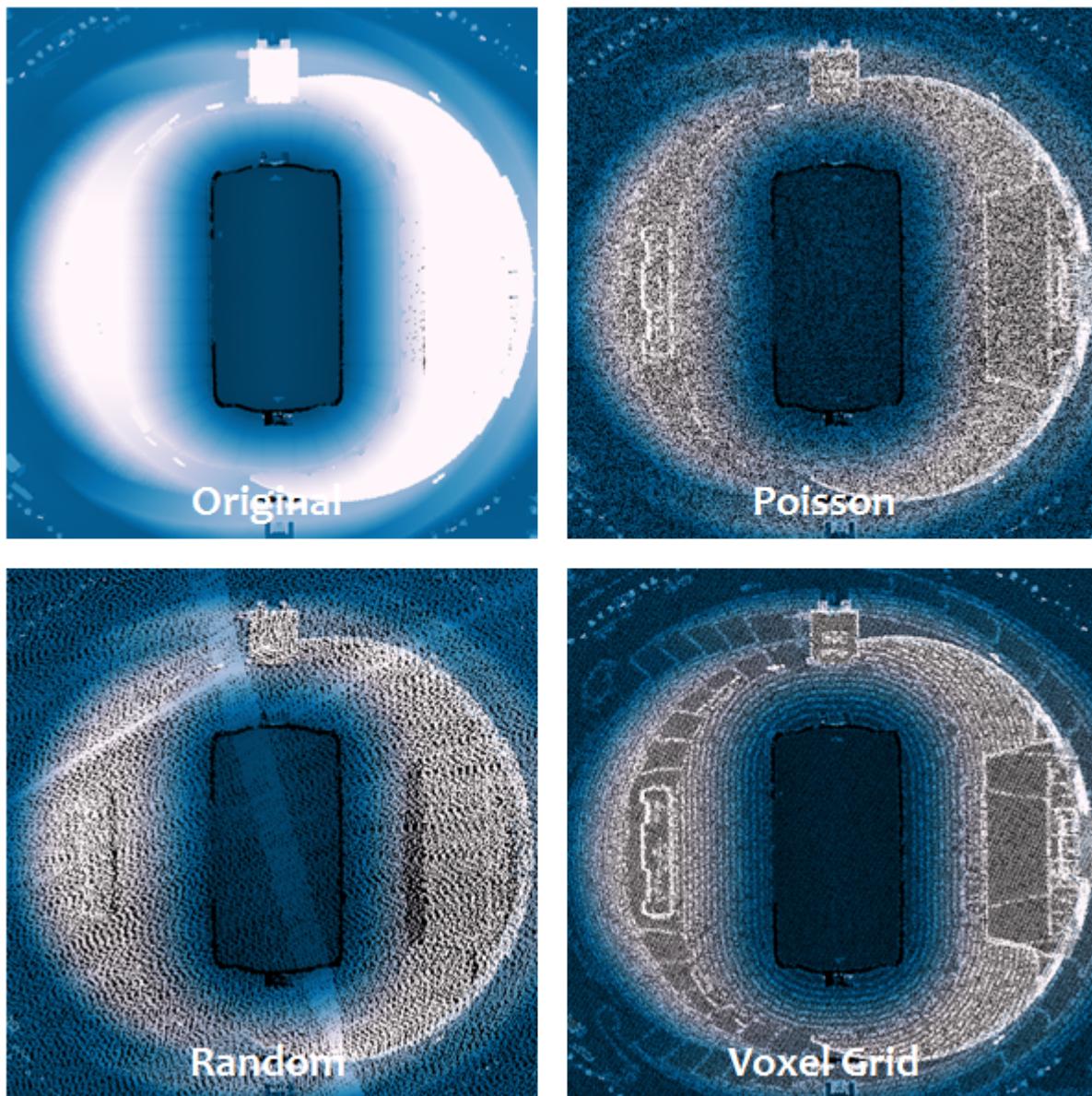
In the next example, we can see that there is an increased number of points in a scan overlap region. This is not uncommon, as data collectors strive to avoid gaps in coverage and overlap datasets to aid in registration of multiple passes. The appearance of these denser regions can be

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.

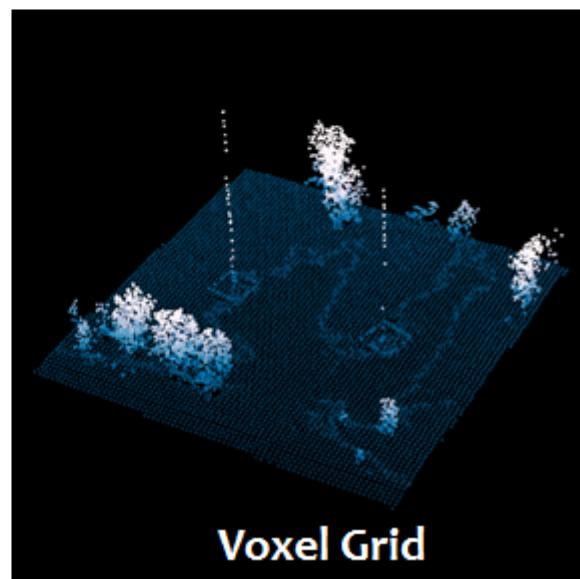
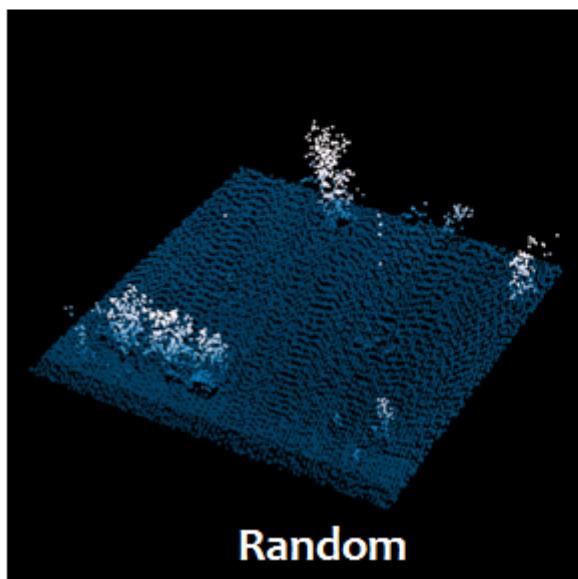
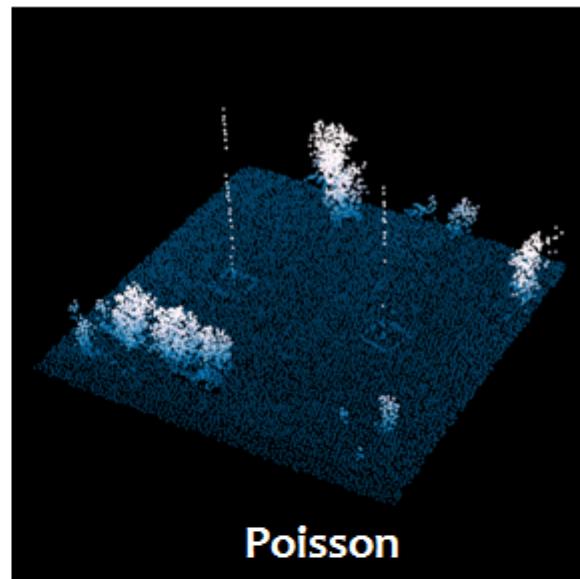
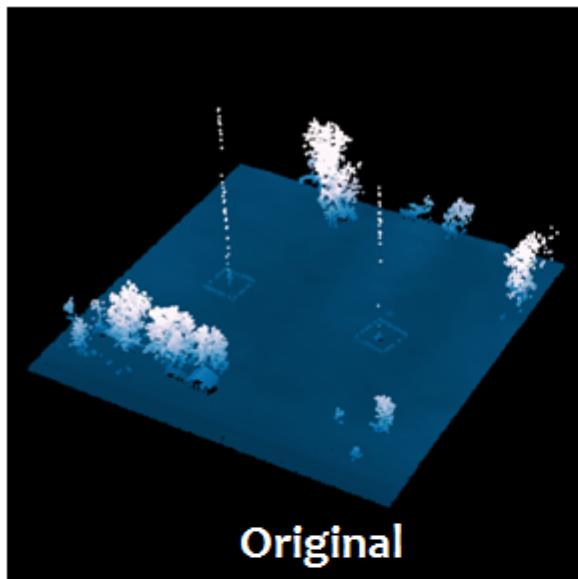


In this top-down view of a football stadium, we again see that the random sampling technique preserves (and perhaps even accentuates) scan pattern and overlap artifacts. It also introduces a side effect to the voxel grid approach, an aliasing of the data, seen as staircasing in the sloping

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.



Developing

Writing a filter

Author Bradley Chambers

Contact brad.chambers@gmail.com

Date 10/26/2016

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 168).

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 120), 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 144), 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, `getDefaultOptions()` 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);
```

Writing a kernel

Author Bradley Chambers

Contact brad.chambers@gmail.com

Date 10/16/2016

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/KernelFactory.hpp>
8 #include <pdal/Options.hpp>
9 #include <pdal/pdal_macros.hpp>
10 #include <pdal/StageFactory.hpp>
11 #include <pdal/PointTable.hpp>
12
13 #include <memory>
14 #include <string>
15
16
17 namespace pdal {
18
19     static PluginInfo const s_info {
20         "kernels.mykernel",
21         "MyKernel",
22         "http://link/to/documentation"
23     };
24
25     CREATE_SHARED_PLUGIN(1, 0, MyKernel, Kernel, s_info);
26     std::string MyKernel::getName() const { return s_info.name; }
27
28     MyKernel::MyKernel() : Kernel()
29     {}
30
31     void MyKernel::addSwitches(ProgramArgs& args)
32     {
33         args.add("input,i", "Input filename", m_input_file).
34         setPositional();
35         args.add("output,o", "Output filename", m_output_file).
36         setPositional();
37     }
38
39     int MyKernel::execute()
40     {
```

```

39     PointTable table;
40     StageFactory f;
41
42     Stage& reader = makeReader(m_input_file, "readers.las");
43
44     // Options should be added in the call to makeFilter, makeReader,
45     // or makeWriter so that the system can override them with those
46     // provided on the command line when applicable.
47     Options filterOptions;
48     filterOptions.add("step", 10);
49     Stage& filter = makeFilter("filters.decimation", reader,
50                               filterOptions);
51
52     Stage& writer = makeWriter(m_output_file, filter, "writers.text");
53
54     writer.prepare(table);
55     writer.execute(table);
56
57     return 0;
58 }
59 } // 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.

```

CREATE_SHARED_PLUGIN(1, 0, MyKernel, Kernel, s_info);
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` and the `pdal::StageFactory`. The latter is used to create the various stages that will be used within the kernel.

```

PointTable table;
StageFactory f;

```

The `pdal::Reader` is created from the `pdal::StageFactory`, 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.

```
Stage& reader = makeReader(m_input_file, "readers.las");

// Options should be added in the call to makeFilter, makeReader,
// or makeWriter so that the system can override them with those
```

The `pdal::Filter` is also created from the `pdal::StageFactory`. 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.

```
Options filterOptions;
filterOptions.add("step", 10);
Stage& filter = makeFilter("filters.decimation", reader,
                           filterOptions);

Stage& writer = makeWriter(m_output_file, filter, "writers.text"
                           );
```

Finally, the `pdal::Writer` is created from the `pdal::StageFactory`. This `writers.text` (page 100), takes as input the previous stage (the `filters.decimation` (page 116)) and the output filename as its sole option.

```
writer.execute(table);

return 0;
}
```

The final two steps are to prepare and execute the pipeline. This is achieved by calling `prepare` and `execute` on the final stage.

```
} // namespace pdal
```

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>.

Writing a reader

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 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
4
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         static Dimension::IdList getDefaultDimensions();
21 }
```

```
22 private:
23     std::unique_ptr<ILeStream> m_stream;
24     point_count_t m_index;
25     double m_scale_z;
26
27     virtual void addDimensions(PointLayoutPtr layout);
28     virtual void addArgs(ProgramArgs& args);
29     virtual void ready(PointTableRef table);
30     virtual point_count_t read(PointViewPtr view, point_count_t
31     ↵count);
32     virtual void done(PointTableRef table);
33 }
```

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 static Dimension::IdList getDefaultDimensions();
```

getDefaultDimensions returns a list of *Dimensions* (page 161) that the reader provides.

```
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 addDimensions(PointLayoutPtr layout);
2     virtual void addArgs(ProgramArgs& args);
3     virtual void ready(PointTableRef table);
4     virtual point_count_t read(PointViewPtr view, point_count_t
5     ↵count);
      virtual void done(PointTableRef table);
```

Various other override methods for the stage. There are a few others that could be overridden, which will not be discussed in this tutorial.

Note: See `./include/pdal/Reader.hpp` of the source tree for more methods that a reader can override or implement.

The source

Again, we start with a full listing of the reader source.

```
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", Dimension::Type::
29     ↪Unsigned64);
30 }
31 Dimension::IdList MyReader::getDefaultDimensions()
32 {
33     Dimension::IdList ids;
34
35     ids.push_back(Dimension::Id::X);
36     ids.push_back(Dimension::Id::Y);
37     ids.push_back(Dimension::Id::Z);
38
39     return ids;
40 }
41
42 void MyReader::ready(PointTableRef)
43 {
44     SpatialReference ref("EPSG:4385");
45     setSpatialReference(ref);
46 }
47
48
49 template <typename T>
50 T convert(const StringList& s, const std::string& name, size_t
51 ↪fieldno)
52 {
53     T output;
54     bool bConverted = Utils::fromString(s[fieldno], output);
55     if (!bConverted)
56     {
57         std::stringstream oss;
58         oss << "Unable to convert " << name << ", " << s[fieldno] <
59         ↪< ", to double";
60         throw pdal_error(oss.str());
61     }
62
63     return output;
64 }
```

```

65 point_count_t MyReader::read(PointViewPtr view, point_count_t _  

66   count)  

67 {  

68     PointLayoutPtr layout = view->layout();  

69     PointId nextId = view->size();  

70     PointId idx = m_index;  

71     point_count_t numRead = 0;  

72  

73     m_stream.reset(new ILeStream(m_filename));  

74  

75     size_t HEADERSIZE(1);  

76     size_t skip_lines(std::max(HEADERSIZE, (size_t)m_index));  

77     size_t line_no(1);  

78     for (std::string line; std::getline(*m_stream->stream(), line); _  

79       line_no++)  

80     {  

81       if (line_no <= skip_lines)  

82       {  

83         continue;  

84       }  

85  

86       // MyReader format: X::Y::Z::Data  

87       StringList s = Utils::split2(line, ':' );  

88  

89       unsigned long u64(0);  

90       if (s.size() != 4)  

91       {  

92         std::stringstream oss;  

93         oss << "Unable to split proper number of fields. Expected _  

94           , got "  

95         << s.size();  

96         throw pdal_error(oss.str());  

97       }  

98  

99       std::string name("X");  

100      view->setField(Dimension::Id::X, nextId, convert<double>(s, _  

101        name, 0));  

102  

103      name = "Y";  

104      view->setField(Dimension::Id::Y, nextId, convert<double>(s, _  

105        name, 1));

```

```
101
102     name = "Z";
103     double z = convert<double>(s, name, 2) * m_scale_z;
104     view->setField(Dimension::Id::Z, nextId, z);
105
106     name = "MyData";
107     view->setField(layout->findProprietaryDim(name),
108                     nextId,
109                     convert<unsigned int>(s, name, 3));
110
111     nextId++;
112     if (m_cb)
113         m_cb(*view, nextId);
114     }
115     m_index = nextId;
116     numRead = nextId;
117
118     return numRead;
119 }
120
121 void MyReader::done(PointTableRef)
122 {
123     m_stream.reset();
124 }
125
126 } //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", Dimension::Type::
7     ↪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 Dimension::IdList MyReader::getDefaultDimensions()
2 {
3     Dimension::IdList ids;
4
5     ids.push_back(Dimension::Id::X);
6     ids.push_back(Dimension::Id::Y);
7     ids.push_back(Dimension::Id::Z);
8
9     return ids;
10 }
```

This method returns the list of *Dimensions* (page 161) that the reader can provide.

```

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 StringList& s, const std::string& name, size_t  
3     ↵fieldno)  
4     {  
5         T output;  
6         bool bConverted = Utils::fromString(s[fieldno], output);  
7         if (!bConverted)  
8             {  
9                 std::stringstream oss;  
10                oss << "Unable to convert " << name << ", " << s[fieldno] <  
11                ↵< ", to double";  
12                throw pdal_error(oss.str());  
13            }  
14        }  
15  
16        return output;  
17    }
```

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  
2     ↵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); ↳line_no++)
2   {
3       if (line_no <= skip_lines)
4       {
5           continue;
6       }

```

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, got "
8           << s.size();
9       throw pdal_error(oss.str());
10  }

```

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";

```

```
8     double z = convert<double>(s, name, 2) * m_scale_z;
9     view->setField(Dimension::Id::Z, nextId, z);
10
11    name = "MyData";
12    view->setField(layout->findProprietaryDim(name),
13                     nextId,
14                     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 1.0.0 REQUIRED CONFIG)
5
```

```
6 set(CMAKE_CXX_FLAGS "-std=c++11")
7 add_library(pdal_plugin_reader_myreader SHARED MyReader.cpp)
8 target_link_libraries(pdal_plugin_reader_myreader PRIVATE ${PDAL_
  ↪LIBRARIES})
9 target_include_directories(pdal_plugin_reader_myreader PRIVATE
  ${PDAL_INCLUDE_DIRS}
10   /Users/chambbj/loki/pdal/repo/readers)
```

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.

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 header

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     static PluginInfo const s_info = PluginInfo(
11         "writers.mywriter",
12         "My Awesome Writer",
13         "http://path/to/documentation" );
14
15     CREATE_SHARED_PLUGIN(1, 0, MyWriter, Writer, s_info);
16
17     std::string MyWriter::getName() const { return s_info.name; }
18
19     struct FileStreamDeleter
20     {
21         template <typename T>
22         void operator()(T* ptr)
23         {
24             if (ptr)
25             {
26                 ptr->flush();
27                 FileUtils::closeFile(ptr);
28             }
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();
```

```

37     args.add("newline", "Line terminator", m_newline, "\n");
38     args.add("datafield", "Data field", m_datafield, "UserData");
39     args.add("precision", "Precision", m_precision, 3);
40 }
41
42 void MyWriter::initialize()
43 {
44     m_stream = FileStreamPtr(FileUtils::createFile(m_filename, true),
45         FileStreamDeleter());
46     if (!m_stream)
47     {
48         std::stringstream out;
49         out << "writers.mywriter couldn't open '" << m_filename <<
50             "' 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
88         *m_stream << x << ":" << y << ":" << z << ":" <<
89         myData << m_newline;
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);
}

```

This method defines the arguments the writer provides and binds them to private variables.

```

{
    std::stringstream out;
    out << "writers.mywriter couldn't open '" << m_filename <<
        "' for output.";
    throw pdal_error(out.str());
}

void MyWriter::ready (PointTableRef table)
{
    m_stream->precision(m_precision);
}

```

```
*m_stream << std::fixed;

Dimension::Id d = table.layout()->findDim(m_datafield);
if (d == Dimension::Id::Unknown)
{
    std::ostringstream oss;
```

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 1.0.0 REQUIRED CONFIG)
5
6 set(CMAKE_CXX_FLAGS "-std=c++11")
7 add_library(pdal_plugin_writer_mywriter SHARED MyWriter.cpp)
8 target_link_libraries(pdal_plugin_writer_mywriter PRIVATE ${PDAL_
  ↴LIBRARIES})
9 target_include_directories(pdal_plugin_writer_mywriter PRIVATE
10   ${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.

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/libblas.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.

**CHAPTER
EIGHT**

WORKSHOP

Point Cloud Processing and Analysis with PDAL

Author Howard Butler

Author Pete Gadomski

Author Dr. Craig Glennie

Contact howard@hobu.co

Date 03/30/2016

Introduction

1. *Introduction to LiDAR* (page 267)
2. *Introduction to PDAL* (page 273)
3. *Software Installation* (page 280)
4. *Basic Information* (page 292)
5. *Translation* (page 299)
6. *Analysis* (page 306)
7. *Georeferencing* (page 360)

Materials

Slides

- Slides (<http://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 (<http://pdal.io/PDAL.pdf>)
- Website (<http://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>

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 431):

$$\mathbf{p}_G^l = \mathbf{p}_{GPS}^l + \mathbf{R}_b^l (\mathbf{R}_s^b \mathbf{r}^s - \mathbf{l}^b) \quad (8.1)$$

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

- R_b^l is the rotation matrix from the navigation frame to the global reference frame
- R_s^b is the rotation matrix from the scanner's frame to the navigation frame (boresight matrix)
- r^s is the coordinates of the laser point in the scanner's frame
- l^b is the lever-arm offset between the scanner's original and the navigation's origin

This equation contains fourteen unknowns, and in order to georeference a single LiDAR return we must determine all fourteen variables at the time of the pulse.

As a rule of thumb, the position, attitude, and motion of the scanning platform (aircraft, vehicle, etc) are sampled at a much lower rate than the pulse rate of the laser — rates of ~1Hz are common for GNSS/IMU sampling. In order to match the GNSS/IMU sampling rate with the sampling rate of the laser, GNSS/IMU measurements are interpolated to line up with the LiDAR measurements. Then, these positions and attitudes are combined via Equation (8.1) 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.

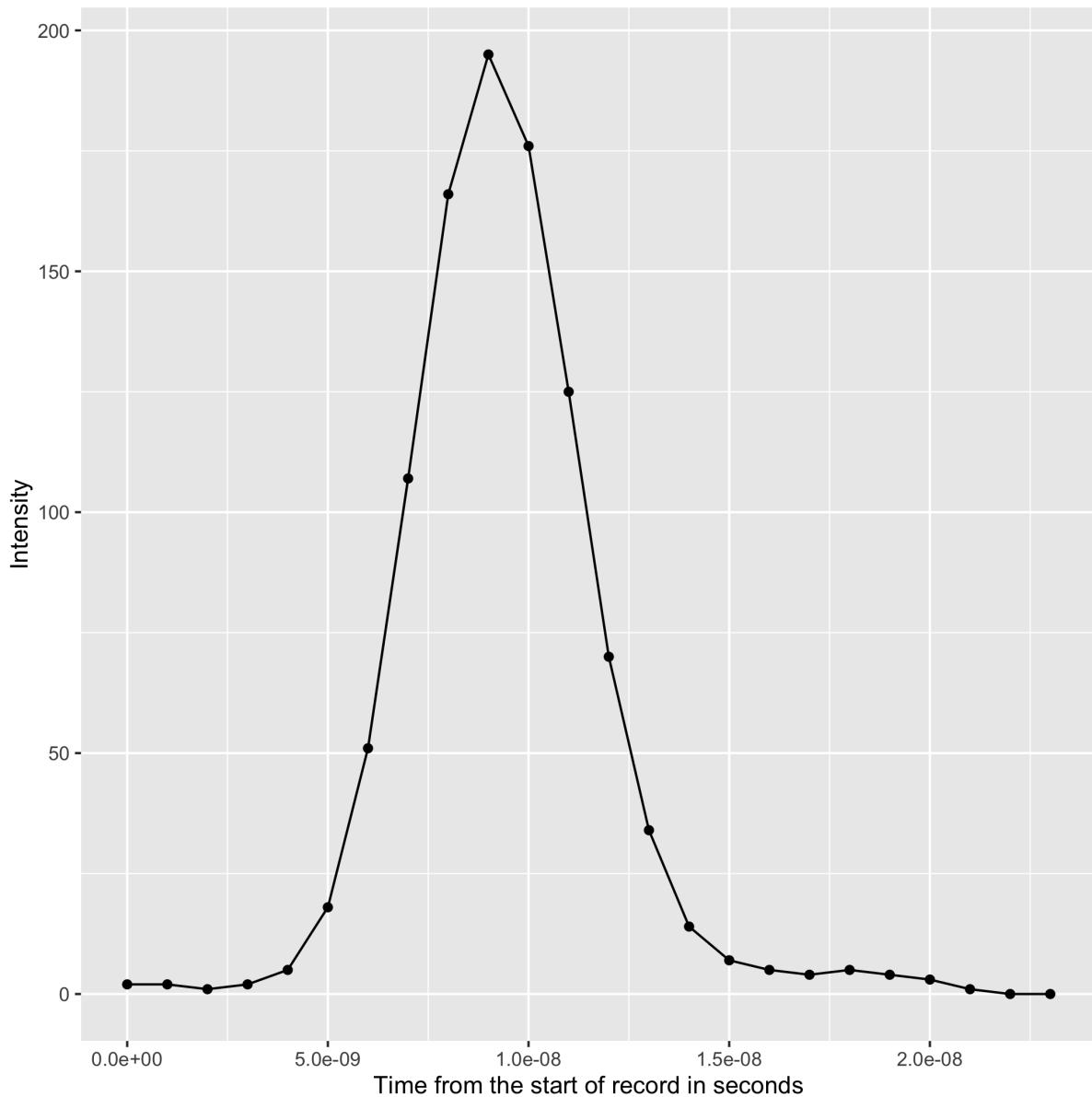


Fig. 8.1: A real-world outgoing LiDAR pulse.

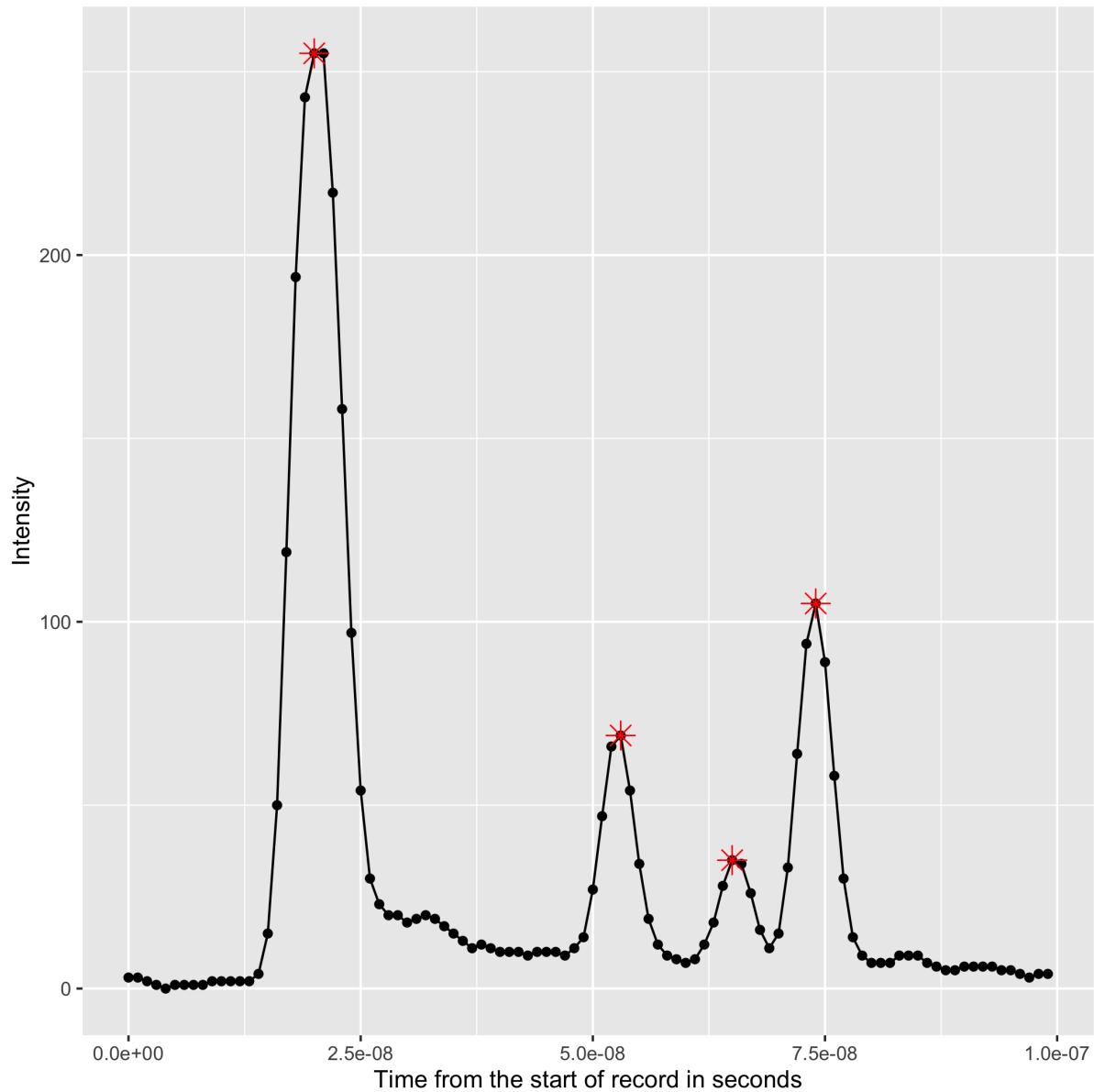


Fig. 8.2: A real-world incoming LiDAR return. Potential discrete-return peaks are marked in red.

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.

Introduction to PDAL

What is PDAL?

PDAL (<http://pdal.io/>) is Point Data Abstraction Library, and it is an open source software for translating and processing point cloud data. It is not limited to just LiDAR (<https://en.wikipedia.org/wiki/Lidar>) data, although the focus and impetus for many of the tools have their origins in LiDAR.

What is its big idea?

Say you wanted to load some [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>) (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 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. It is a kind of one-off, and it could quickly spiral out of control as you look to 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.

Little programs that encapsulate specific functionality that can be composed together provide a more streamlined approach to the problem. They 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 57) stage, a [Reprojection](#) (page 152) stage, and a [PostgreSQL Writer](#) (page 94), 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. 8.3: A simple PDAL pipeline composed of a reader, filter, and writer stages.

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) (<https://en.wikipedia.org/wiki/JSON>) or [XML](https://en.wikipedia.org/wiki/XML) (<https://en.wikipedia.org/wiki/XML>), and it provides a command, [pipeline](#) (page 27), 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/) (<http://gdal.org/>)'s [Virtual Raster Format](http://www.gdal.org/gdal_vrttut.html) (http://www.gdal.org/gdal_vrttut.html).

How is it different than other tools?

LAStools

One of the most common open source processing tool suites available 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 must be able to generically 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 its view of formats it understands is within the context of the dimension types provided by the LAS format.

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 190) for more detail on how to take advantage of PCL capabilities within PDAL operations.

Greyhound and Entwine

[Greyhound](http://github.com/hobu/greyhound) (<http://github.com/hobu/greyhound>) is an open source software from Hobu, Inc. that allows clients over the internet to query and stream progressive point cloud data. [Entwine](http://github.com/connormanning/entwine) (<http://github.com/connormanning/entwine>) is an open source software from Hobu, Inc. that organizes massive point cloud collections into [Greyhound](http://github.com/hobu/greyhound) (<http://github.com/hobu/greyhound>)-streamable data services. These two software projects allow province-scale LiDAR collections to be organized and served via HTTP clients over the internet.

plas.io and Potree

plas.io (<http://plas.io>) is a [WebGL](https://en.wikipedia.org/wiki/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) (<http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html>) and [LASzip](http://laszip.org) (<http://laszip.org>) compressed LAS.

PDAL: Point cloud Data Abstraction Library, 1.4.0

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) (<http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html>) and [LASzip](http://laszip.org) (<http://laszip.org>) compressed LAS.

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

Note: The [libLAS](http://liblas.org) (<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.

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 GRiD (<http://lidar.io/>) project. Functionality that was creeping into libLAS 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.

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.

What are PDAL's weak points?

PDAL doesn't provide a friendly GUI interface, it expects that you have the confidence to dig into a command-line interface, and it sometimes forgets that you don't always want to read source code to figure out what exactly is happening. PDAL is an open source project in active development, and because of that, we're always working to improve it. Please visit [Community](#) (page 35) 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.

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 17) 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 421) 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.

See also:

PDAL [software](#) (page 181) [development](#) (page 185) [tutorials](#) (page 237) have more information on how to use the library from a software developer's perspective. We won't get very deep in the C++ swamp in this workshop.

Command Line Utilities

PDAL provides a number of [applications](#) (page 17) 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 22) about a data set

- Data *translation* (page 32) 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 20)
- *Merge* (page 26) or *split* (page 28) data
- *Catalog* (page 29) collections of data

Python API

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

Embed

By embedding Python, PDAL allows you to interact with point cloud data using typical [Numpy](http://www.numpy.org/) (<http://www.numpy.org/>) features. PDAL embeds [Python](http://python.org/) (<http://python.org/>) scripts in your processing work flows with the *filters.programmable* (page 148) and *filters.predicate* (page 146) filters. Your Python scripts can process and interact with point cloud data during the execution of a *PDAL pipeline* (page 37), and you are free to dynamically do whatever you want in your scripts.

Extension

PDAL also provides a Python extension for software developers who simply want to use data as a mechanism to abstract data formats. This approach works really well in algorithm work bench scenarios, simple data testing and validation challenges, or situations where full C++ applications would be too much effort or complexity.

See also:

The [Python installation](#) (page 395) document contains information on how to install and use the PDAL Python extension.

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.

Software Installation

Docker

What is Docker?

Docker (<https://www.docker.com/>) is operating system virtualization. It is somewhat like a virtual machine, but instead of the machine being abstracted, the operating system is. The advantage it gives you is the ability to run many different things in many different configurations and not have them collide with each other.

How will we use Docker?

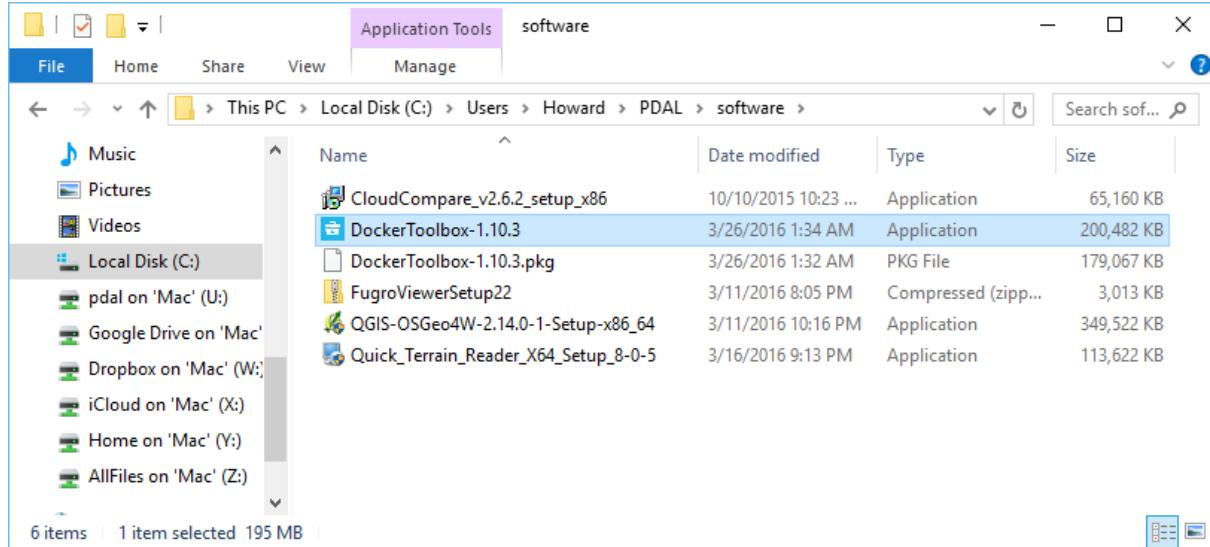
PDAL stands on the shoulders of giants. It uses GDAL, GEOS, and [many other dependencies](#) (page 365). Because of this, it is very challenging to build it yourself. We could easily burn an entire workshop learning the esoteric build miseries of PDAL and all of its dependencies. Fortunately, Docker will allow us to use a fully-featured known configuration to run our examples and exercises without having to suffer so much.

Installing Docker

1. Copy the entire contents of your workshop USB key to a PDAL directory in your home directory (something like C:\Users\Howard\PDAL). We will refer to this location for the rest of the workshop materials.

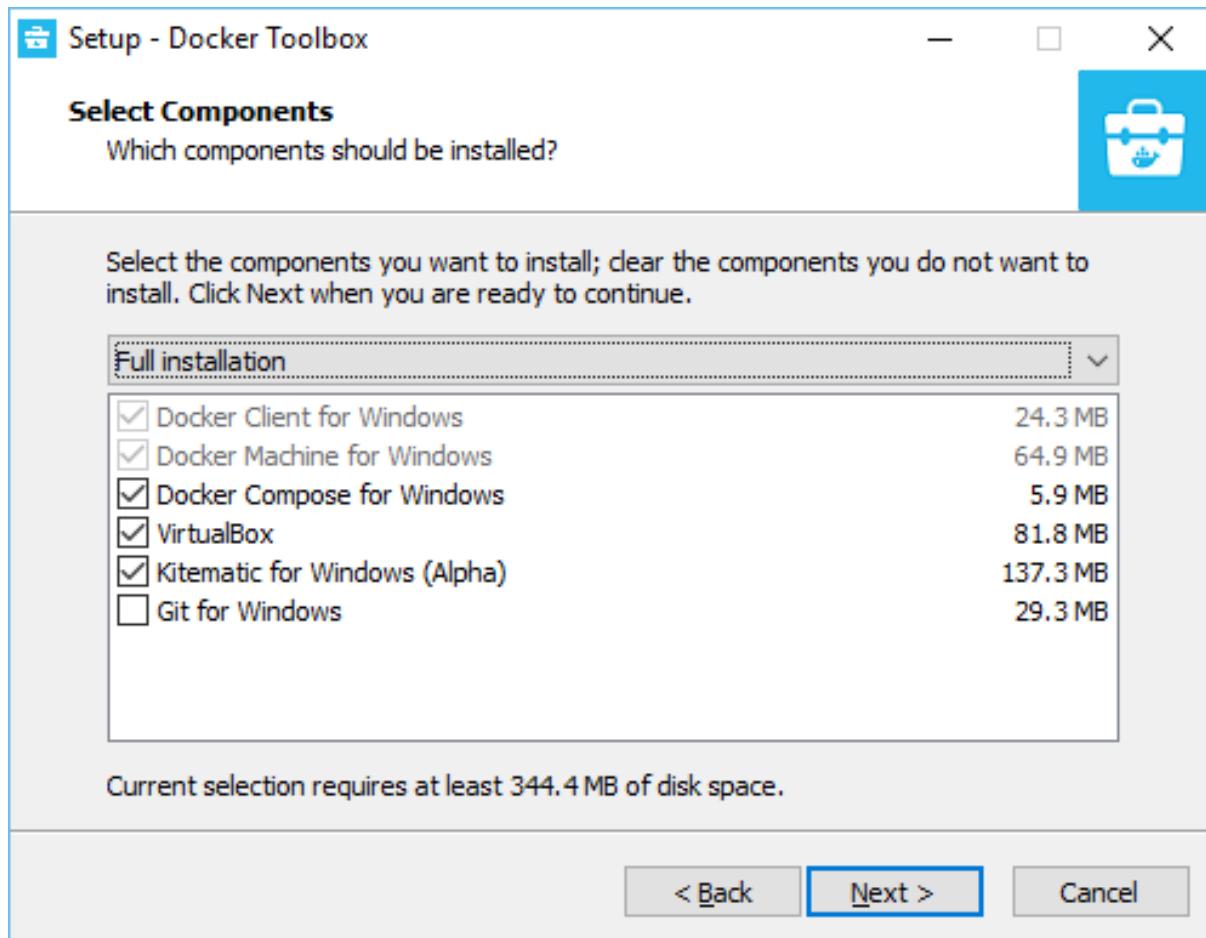
Note: The workshop materials require ~5gb of local disk storage. You might be able to run directly from the USB drive, but this scenario has not been tested.

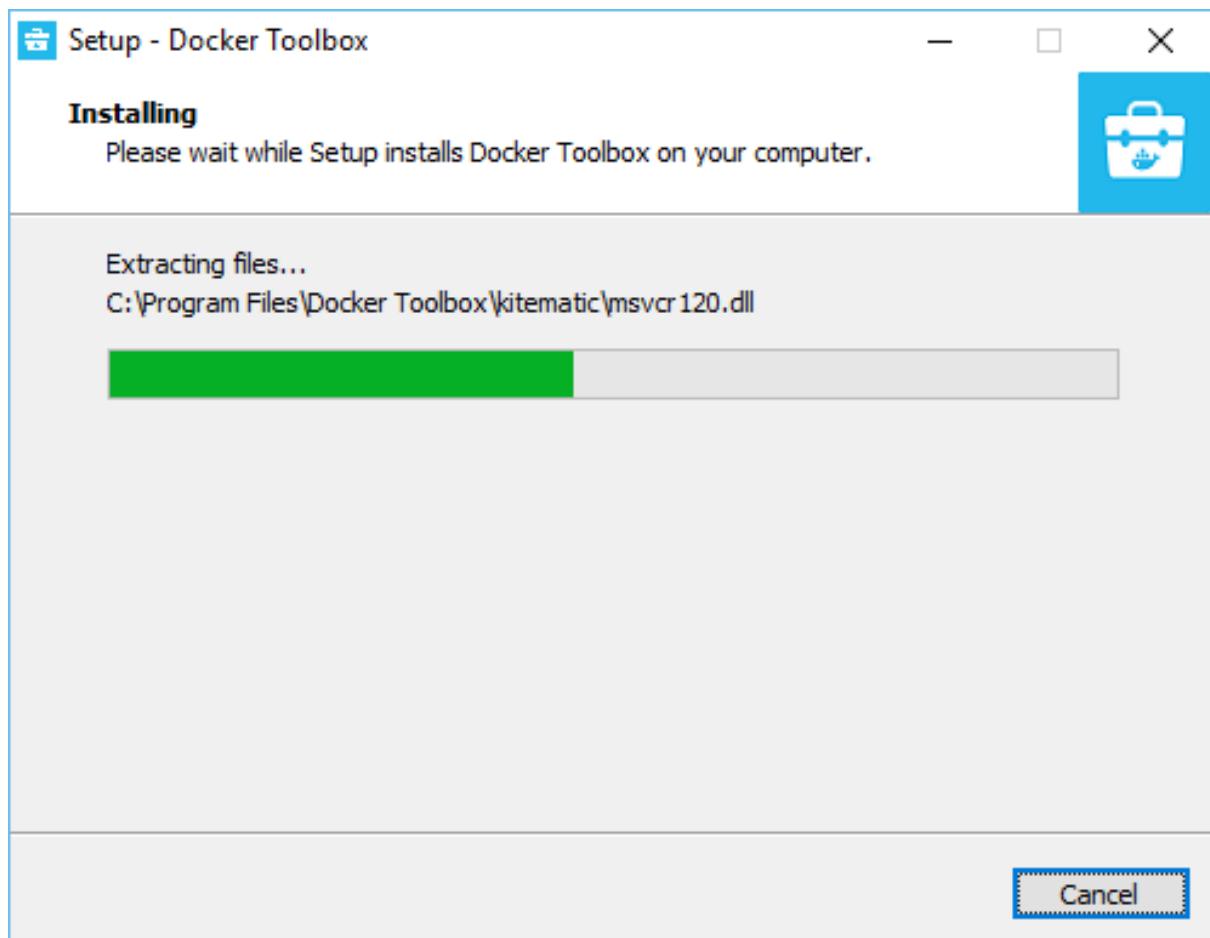
2. After your materials are copied, navigate to the c:\Users\Howard\PDAL\docker directory.



Note: It is assumed your Hobu USB drive has all of its contents copied to the C:\Users\Howard\PDAL folder. Please adjust your locations when reading these tutorial documents accordingly.

3. Choose the install image, Windows or Mac, and install Docker Toolbox (<https://www.docker.com/products/docker-toolbox>) to prepare your machine to run the examples.

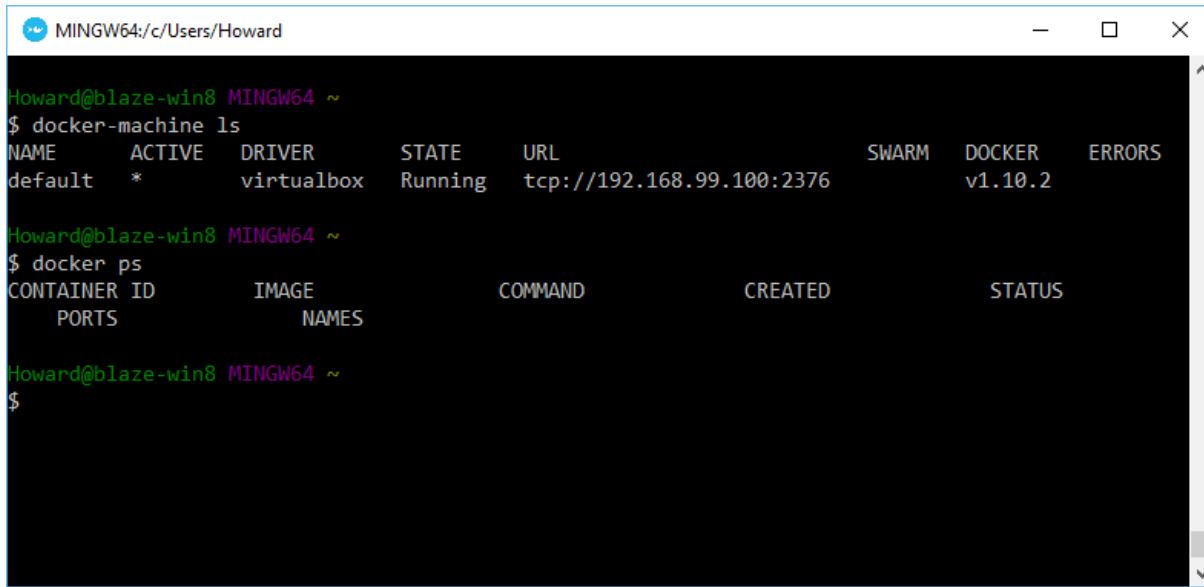




4. Once installed, choose the *Docker Quickstart Terminal* from your Desktop. It will scroll a bunch of text across the terminal screen as it does setup and configuration. Once it is done, verify that things are working correctly by issuing the following commands:

```
$ docker-machine ls
```

```
$ docker ps
```



The screenshot shows a terminal window titled "MINGW64:/c/Users/Howard". It displays the output of several Docker commands:

```
Howard@blaze-win8 MINGW64 ~
$ docker-machine ls
NAME      ACTIVE     DRIVER      STATE      URL
default    *          virtualbox   Running    tcp://192.168.99.100:2376
SWARM      DOCKER     v1.10.2
           ERRORS

Howard@blaze-win8 MINGW64 ~
$ docker ps
CONTAINER ID        IMAGE               COMMAND
PORTS             NAMES
$
```

Images and Containers

There are a few Docker concepts to understand before they are very useful. The first and most important one is the concept of *images*. A Docker image is kind of like an environment that is frozen and not yet running. When you actually run an image, the thing running is called a *container*. Containers are what do the work and virtualize the operating system.

A downside to docker images is they are often really large. Gigabytes. We don't want to swamp the bandwidth here at the workshop venue, so we're going to load images from the Hobu workshop USB drive.

1. Load the PDAL Docker image:

If you have a PDAL workshop USB key, you can load it from your workspace contained on it.

```
$ docker load -i /c/Users/Howard/PDAL/docker/images/
→pdal.tar
```

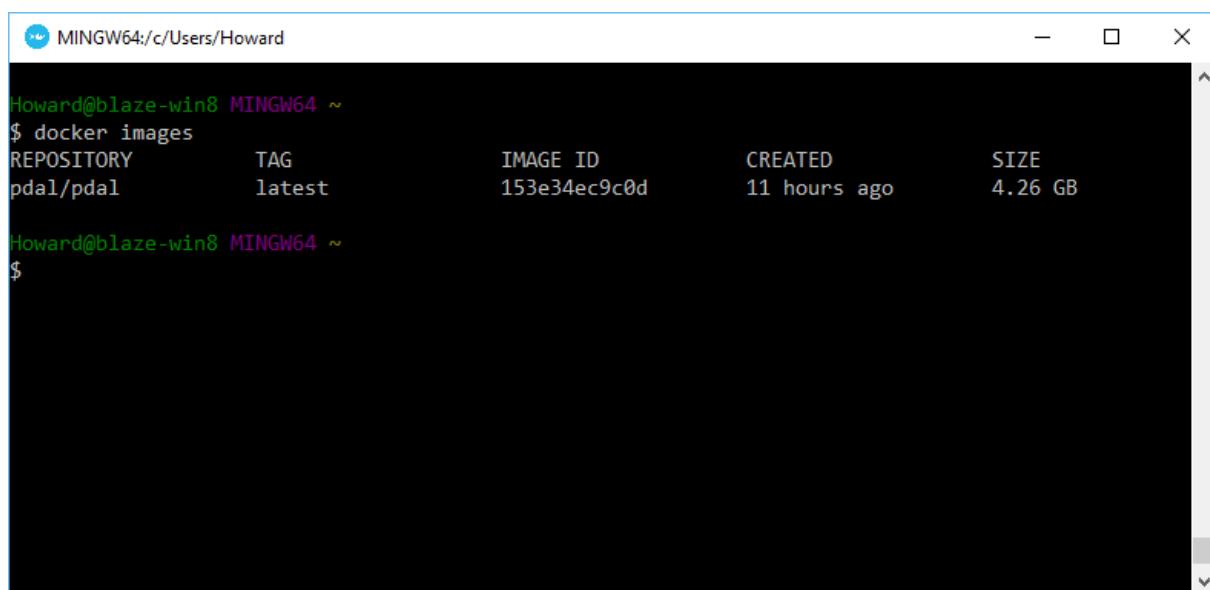
If you got your workshop materials from the internet, you can simply issue a `pull` command and Docker will fetch and load it for you.

```
$ docker pull pdal/pdal
```

Note: The *Docker Quickstart Terminal* is [Bash](https://www.gnu.org/software/bash/) (<https://www.gnu.org/software/bash/>), not normal Windows cmd.exe. Because of this, you will need to use double slashes from time to time when providing file paths.

2. Verify that the image successfully loaded

```
$ docker images
```



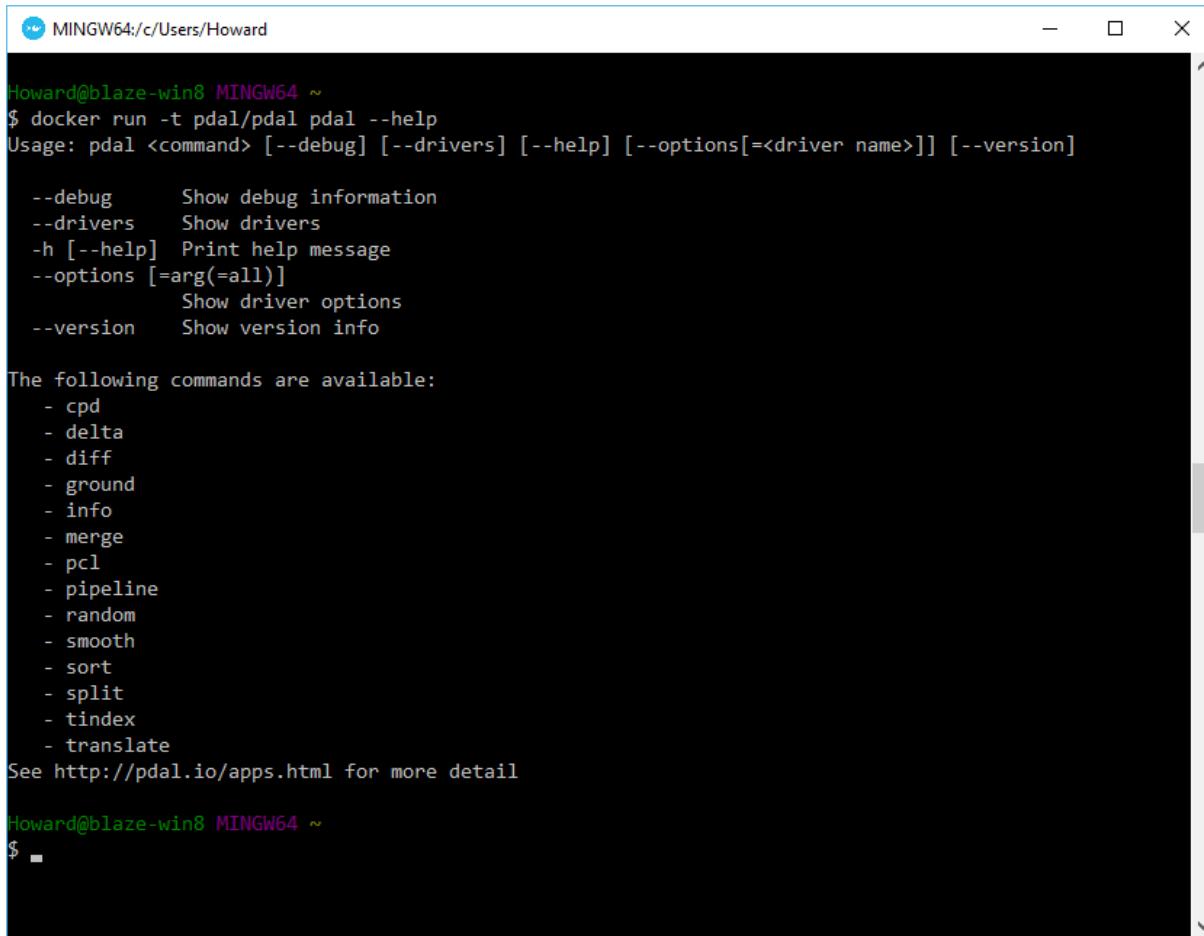
The screenshot shows a terminal window titled 'MINGW64:/c/Users/Howard'. The command '\$ docker images' is run, displaying the following table:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
pdal/pdal	latest	153e34ec9c0d	11 hours ago	4.26 GB

3. Verify PDAL can print help output

```
$ docker run -t pdal/pdal pdal --help
```

PDAL: Point cloud Data Abstraction Library, 1.4.0

A screenshot of a terminal window titled "MINGW64:/c/Users/Howard". The window shows the output of a command to run the PDAL Docker container and print help information. The help text describes various PDAL commands like cpd, delta, diff, ground, info, merge, pcl, pipeline, random, smooth, sort, split, tindex, and translate, along with their descriptions. It also provides links to the PDAL documentation and source code.

```
MINGW64:/c/Users/Howard
Howard@blaze-win8 MINGW64 ~
$ docker run -t pdal/pdal pdal --help
Usage: pdal <command> [--debug] [--drivers] [--help] [--options[=<driver name>]] [--version]

--debug      Show debug information
--drivers    Show drivers
-h [--help]   Print help message
--options [=arg(=all)]
              Show driver options
--version    Show version info

The following commands are available:
- cpd
- delta
- diff
- ground
- info
- merge
- pcl
- pipeline
- random
- smooth
- sort
- split
- tindex
- translate
See http://pdal.io/apps.html for more detail

Howard@blaze-win8 MINGW64 ~
$
```

Conclusion

Once you have verified that you can run the pdal/pdal container and print help information, you are ready to begin the *Exercises* (page 292).

QGIS

What is QGIS (<http://qgis.org>)?

QGIS (<http://qgis.org>) is an open source GIS. It is extensible with Python (<http://python.org/>), it integrates the GRASS (<https://grass.osgeo.org/>) analytic environment, and it works on both

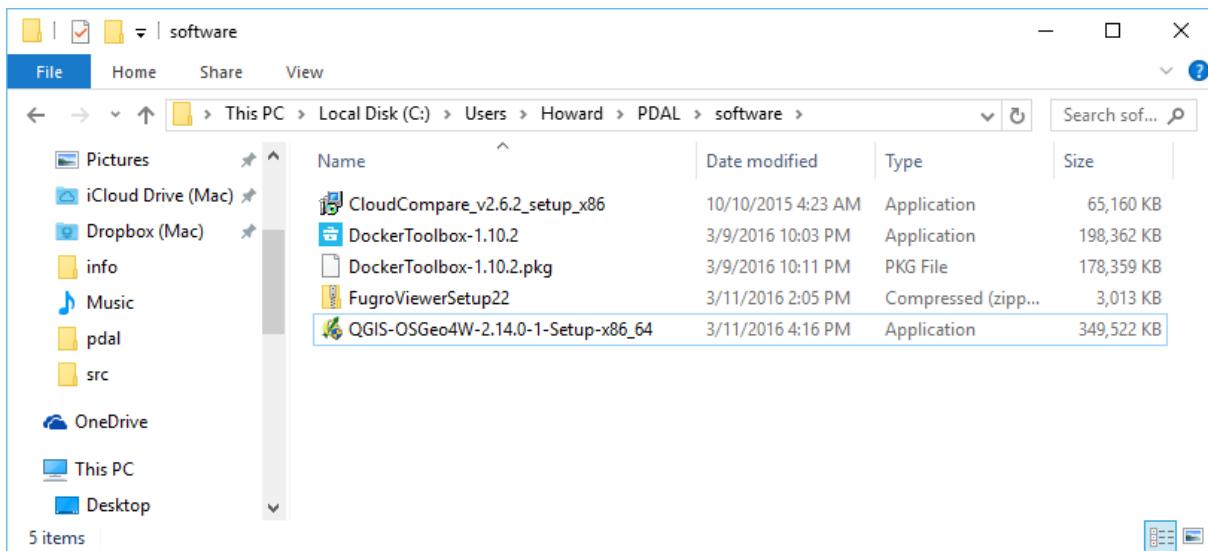
Windows and OS X.

How will we use QGIS?

We're using **QGIS** (<http://qgis.org>) to visualize raster and vector processing product during our workshop. If you have another GIS available to you, you are welcome to use it, but because **QGIS** (<http://qgis.org>) is open source, we are installing it and using it to be sure you'll have something to look at data with.

Installing QGIS

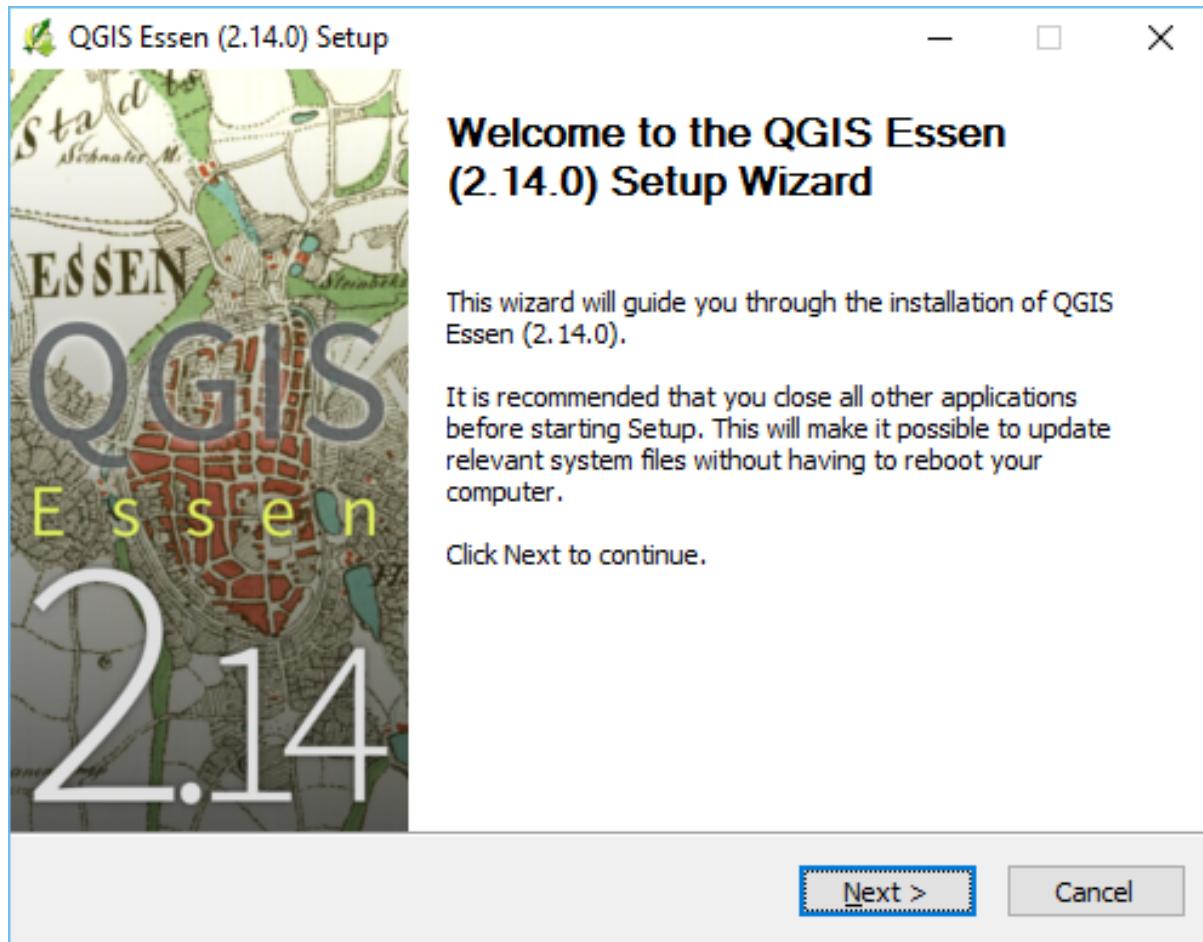
1. Copy the contents of your **Hobu** (<http://hobu.co/>) USB key to a PDAL directory in your home directory (something like C:\Users\Howard\PDAL). We will refer to this location for the rest of the workshop materials.
2. After your materials are copied, navigate to the c:\Users\Howard\PDAL\software directory.

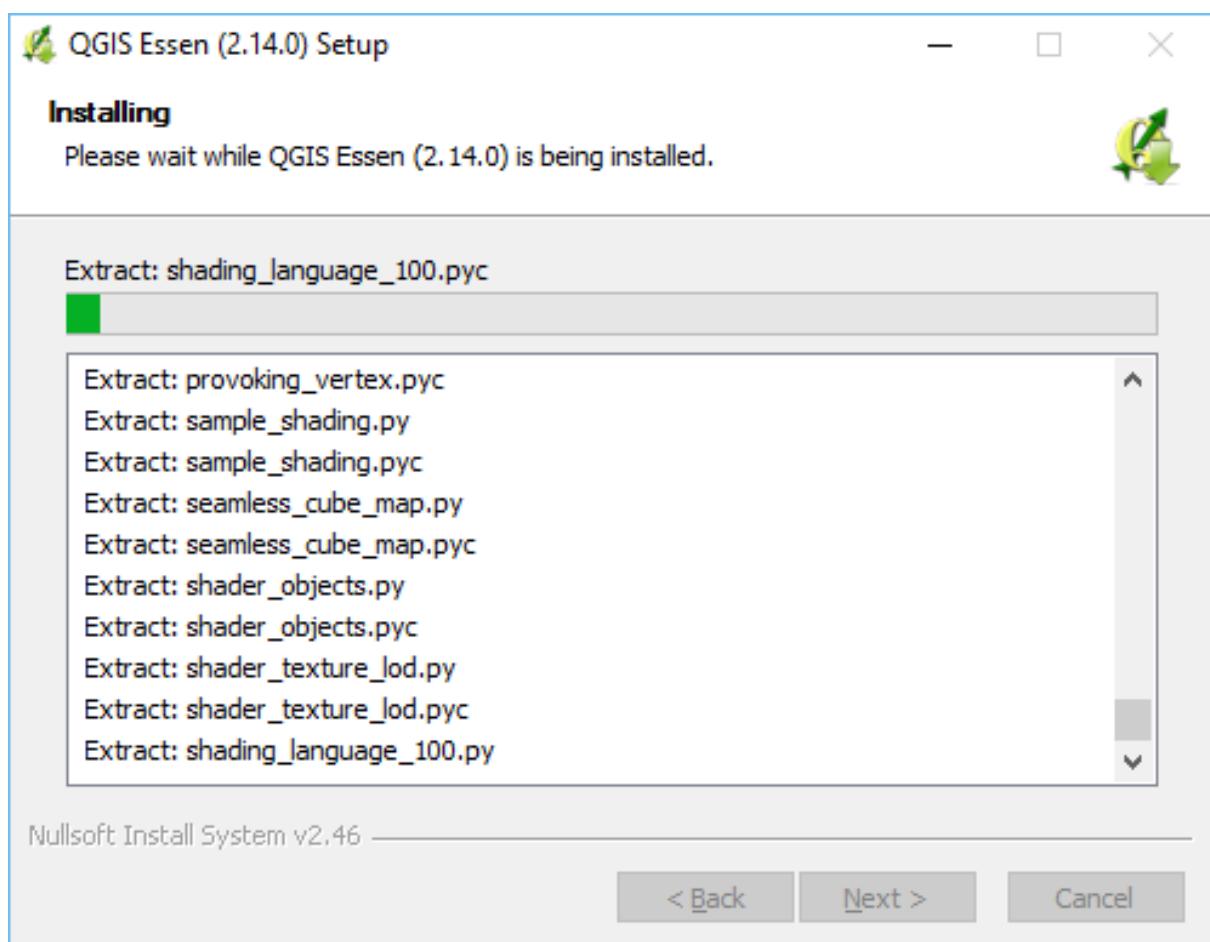


Note: It is assumed your **Hobu** (<http://hobu.co/>) USB drive has all of its contents copied to the C:\Users\Howard\PDAL folder. Please adjust your locations when reading these tutorial

documents accordingly.

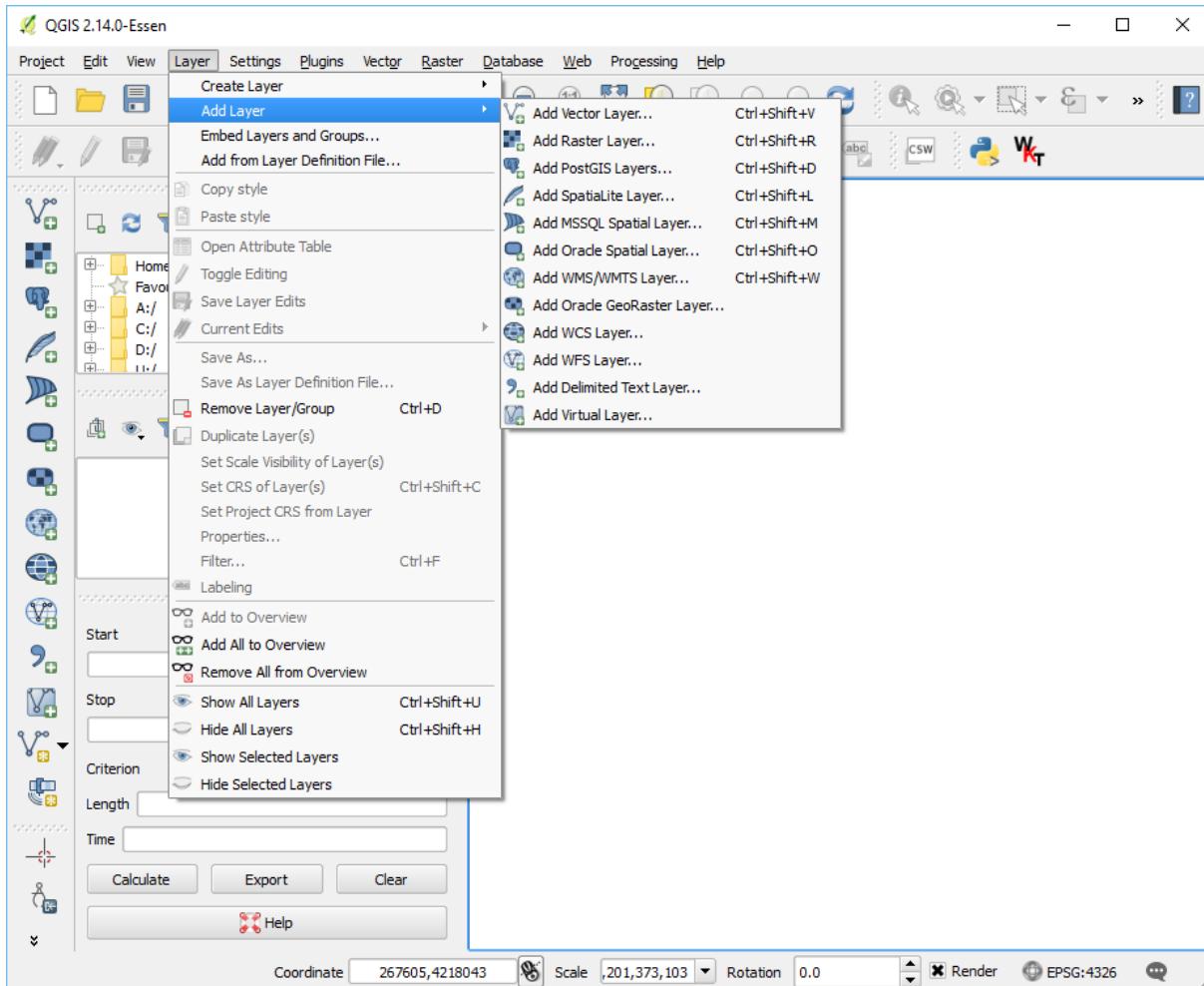
3. Choose the install image, Windows or Mac, and install [QGIS](http://qgis.org) (<http://qgis.org>) prepare your machine to run the examples.



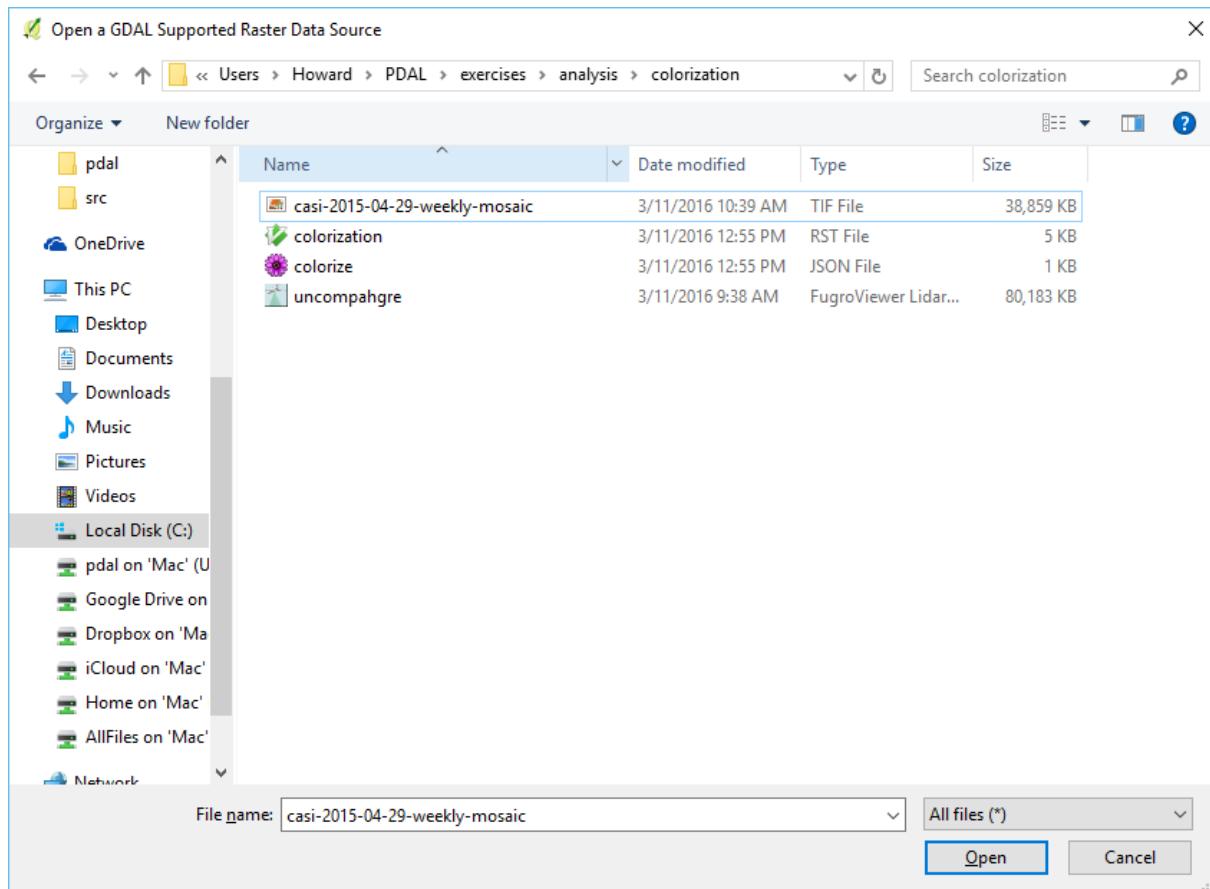


4. Once installed, verify you can run [QGIS](http://qgis.org) (<http://qgis.org>) by opening the application.
Navigate to the
C:\Users\Howard\PDAL\exercises\analysis\colorization directory.

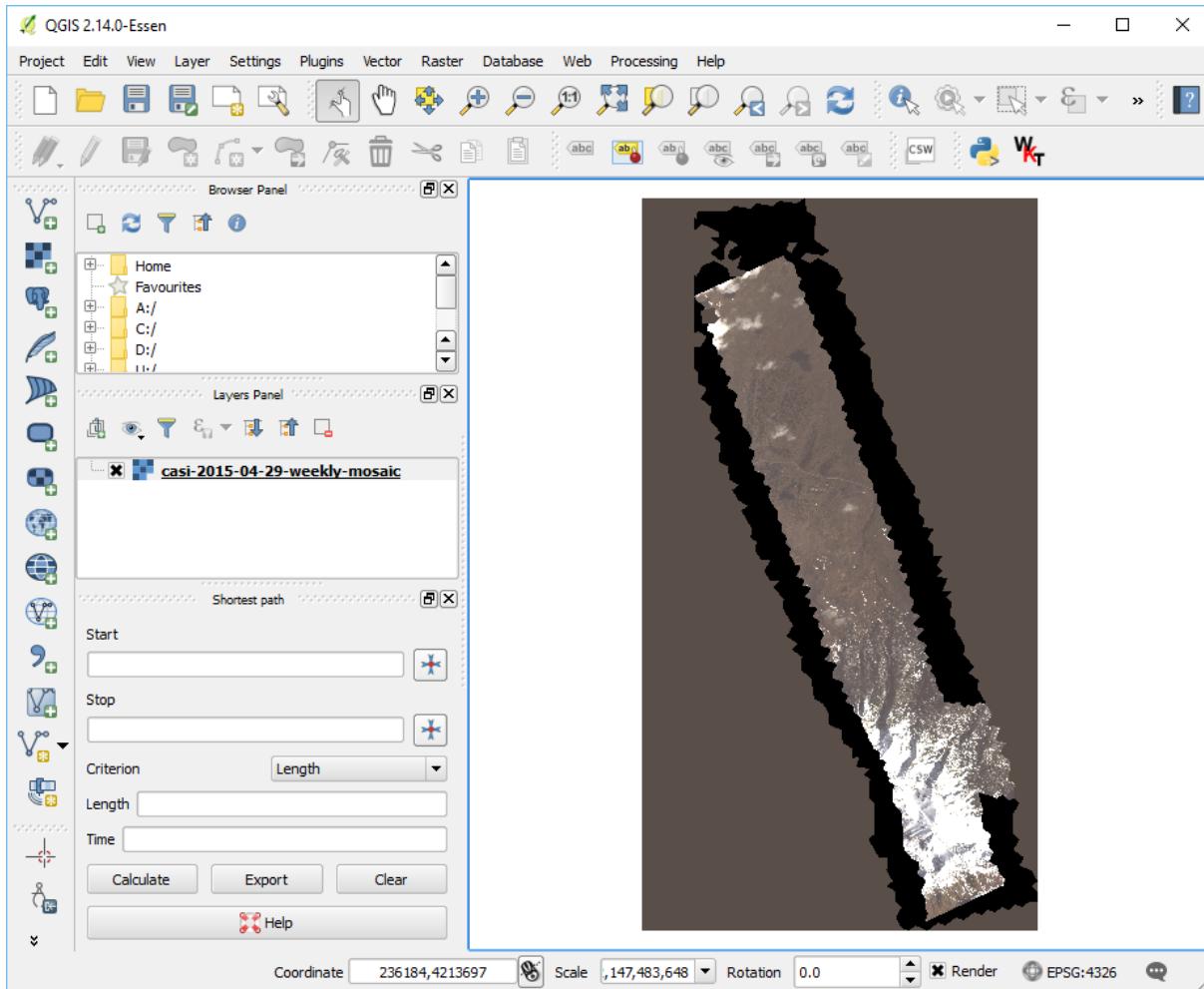
PDAL: Point cloud Data Abstraction Library, 1.4.0



5. Select the `casi-2015-04-29-weekly-mosaic.tif` image and open it for display.



PDAL: Point cloud Data Abstraction Library, 1.4.0



Conclusion

QGIS (<http://qgis.org>) allows everyone to have access to a fully-featured GIS. We are going to use it to visualize raster and vector data used throughout the workshop.

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 *Docker Quickstart Terminal*.

```
1 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
2     pdal info /data/exercises/info/interesting.las -p 0
```

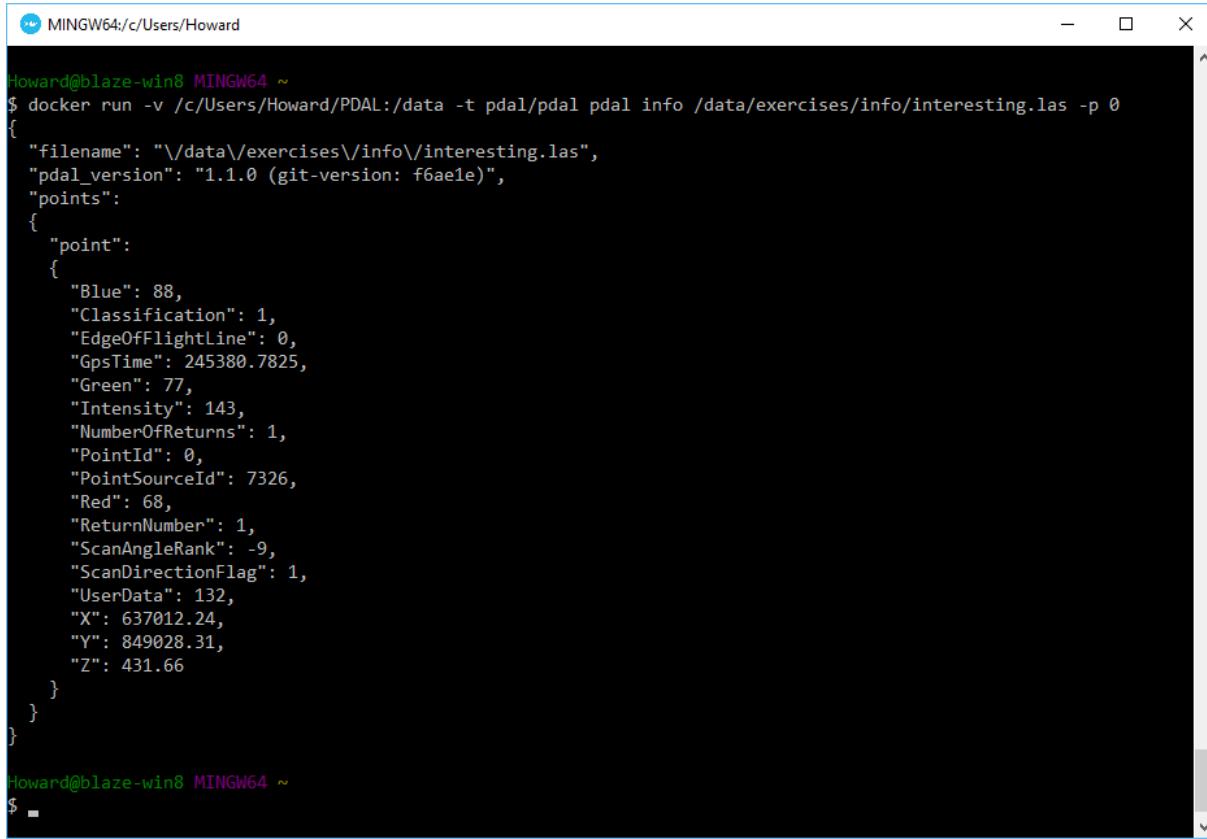
Here's a summary of what's going on with that command invocation

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/Howard/PDAL:/data`: Maps our workshop 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`: This is the Docker image we are going to run. We fetched it in the [Install Docker](#) (page 9) portion of the workshop.
5. `pdal`: We're finally going to run the `pdal` application :)
6. `info`: We want to run [info](#) (page 22) on the data. All commands are run by the `pdal` application.
7. `/data/exercises/info/interesting.las`: 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 `interesting.las` file resides there.
8. `-p 0`: `-p` corresponds to “print a point”, and 0 means to print the first one (computer people count from 0).

A screenshot of a terminal window titled "MINGW64:/c/Users/Howard". The window shows a command being run: "docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal pdal info /data/exercises/info/interesting.las -p 0". The output is a JSON object describing a point from a LAS file. The point has attributes like "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, and "Z": 431.66.

```
MINGW64:/c/Users/Howard
Howard@blaze-win8 MINGW64 ~
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal pdal info /data/exercises/info/interesting.las -p 0
{
  "filename": "\/data\/exercises\/info\/interesting.las",
  "pdal_version": "1.1.0 (git-version: f6ae1e)",
  "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
      }
    }
  ]
}
Howard@blaze-win8 MINGW64 ~
$
```

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 22). 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. You can use the [writers.text](#) (page 100) 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.
5. In the command, we add the `\` character for line continuation. All items on the `docker run` command must be on the same line. You will see this convention throughout the

workshop to make the command easier to read, but remember that everything needs to be on one line.

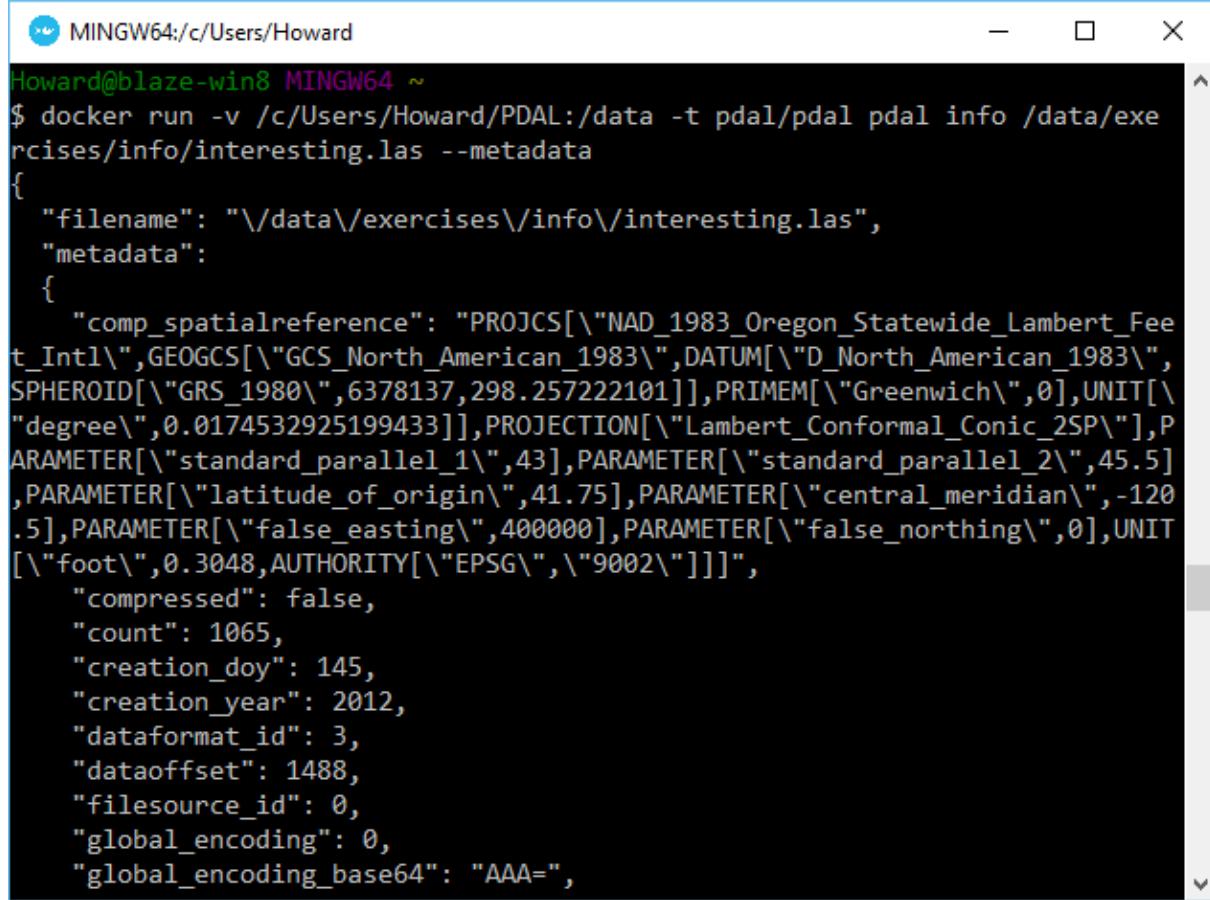
Printing file metadata

Exercise

This exercise uses PDAL to print metadata information. Issue the following command in your *Docker Quickstart Terminal*.

literalinclude:: ./metadata-command.txt

linenos

A screenshot of a terminal window titled "MINGW64:/c/Users/Howard". The command \$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal pdal info /data/exercises/info/interesting.las --metadata is run. The output shows detailed metadata for the point cloud, including its spatial reference system (NAD_1983_Oregon_Statewide_Lambert_Fee_t_Intl), coordinate system (GCS_North_American_1983), datum (D_North_American_1983), spheroid (GRS_1980), primem (Greenwich), unit (degree), projection (Lambert_Conformal_Conic_2SP), and various parameters like standard_parallel, latitude_of_origin, central_meridian, false_easting, and false_northing. It also includes compressed status, count, creation_doy, creation_year, dataformat_id, dataoffset, filesource_id, global_encoding, and global_encoding_base64.

```
MINGW64:/c/Users/Howard
Howard@blaze-win8 MINGW64 ~
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal pdal info /data/exercises/info/interesting.las --metadata
{
  "filename": "\/data\/exercises\/info\/interesting.las",
  "metadata": {
    "comp_spatialreference": "PROJCS[\"NAD_1983_Oregon_Statewide_Lambert_Fee_t_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=",
```

Note: PDAL *metadata* (page 403) is returned as 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)
- [Ruby JSON library](http://ruby-doc.org/stdlib-2.0.0/libdoc/json/rdoc/JSON.html) (<http://ruby-doc.org/stdlib-2.0.0/libdoc/json/rdoc/JSON.html>)

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 22). 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 403) 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 48) produce same-named values where possible, but it is common that variables are different. *Filters* (page 101) and even *writers* (page 74) 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.

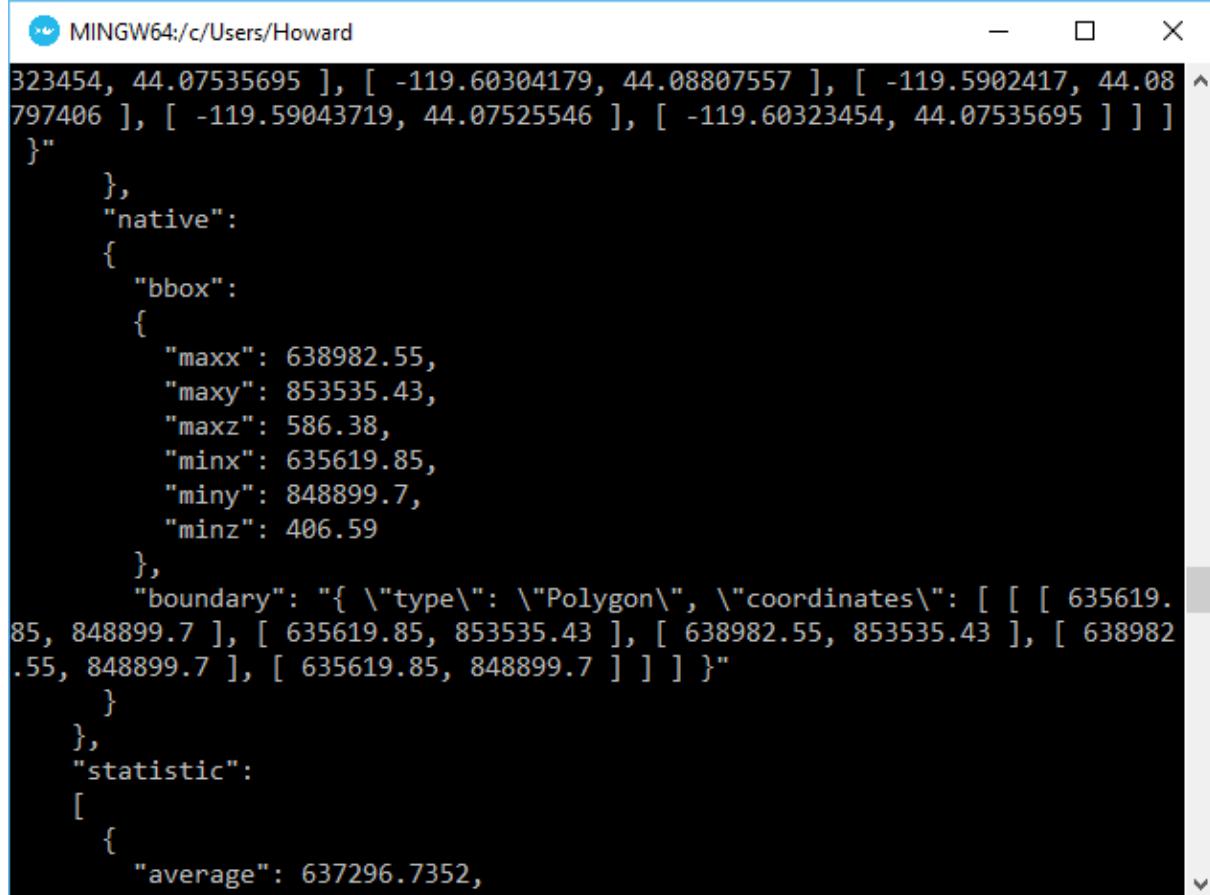
Searching near a point

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:

```
docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
pdal info /data/exercises/info/interesting.las --all
```



```
323454, 44.07535695 ], [ -119.60304179, 44.08807557 ], [ -119.5902417, 44.08
797406 ], [ -119.59043719, 44.07525546 ], [ -119.60323454, 44.07535695 ] ] ]
}"
},
"native":
{
  "bbox":
  {
    "maxx": 638982.55,
    "maxy": 853535.43,
    "maxz": 586.38,
    "minx": 635619.85,
    "miny": 848899.7,
    "minz": 406.59
  },
  "boundary": "{ \"type\": \"Polygon\", \"coordinates\": [ [ [ 635619.
85, 848899.7 ], [ 635619.85, 853535.43 ], [ 638982.55, 853535.43 ], [ 638982
.55, 848899.7 ], [ 635619.85, 848899.7 ] ] ] }"
}
},
"statistic":
[
  {
    "average": 637296.7352,
```

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

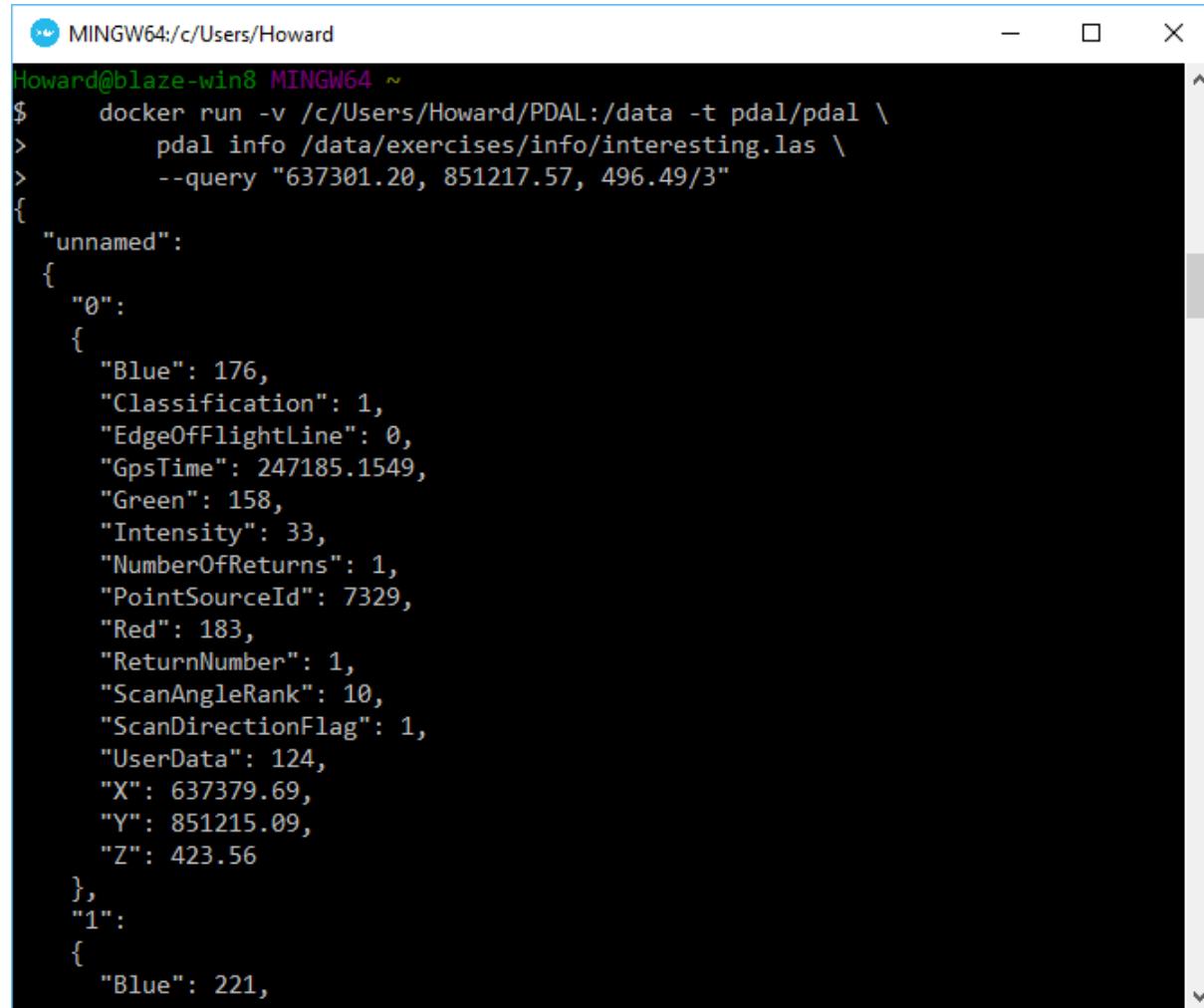
```
x = 635619.85 + (638982.55 - 635619.85)/2 = 637301.20
y = 848899.70 + (853535.43 - 848899.70)/2 = 851217.57
z = 406.59 + (586.38 - 406.59)/2 = 496.49
```

With our “center point”, issue the --query option to pdal info and return the three nearest points to it:

PDAL: Point cloud Data Abstraction Library, 1.4.0

```
docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
pdal info /data/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 terminal window titled 'MINGW64:/c/Users/Howard'. The command entered is:

```
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
pdal info /data/exercises/info/interesting.las \
--query "637301.20, 851217.57, 496.49/3"
```

The output is a JSON object representing a point with three nearest neighbors:{
 "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>) as the exchange format when printing information from **info** (page 22). JSON is a structured, human-readable format that is much simpler than its **XML** (<https://en.wikipedia.org/wiki/XML>) cousin.
2. The **--query** option of **info** (page 22) constructs a **KD-tree** (https://en.wikipedia.org/wiki/K-d_tree) of the entire set of points in memory. If you have really large data sets, this isn't going to work so well, and you will need to come up with a different solution.

Translation

Compression

Exercise

This exercise uses PDAL to compress **ASPRS LAS** (<http://www.asprs.org/Committee-General/LASer-LAS-File-Format-Exchange-Activities.html>) data into **LASzip** (<http://laszip.org>).

Issue the following command in your *Docker Quickstart Terminal*.

```
docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
    pdal translate /data/exercises/translation/interesting.las \
    /data/exercises/translation/interesting.laz
```

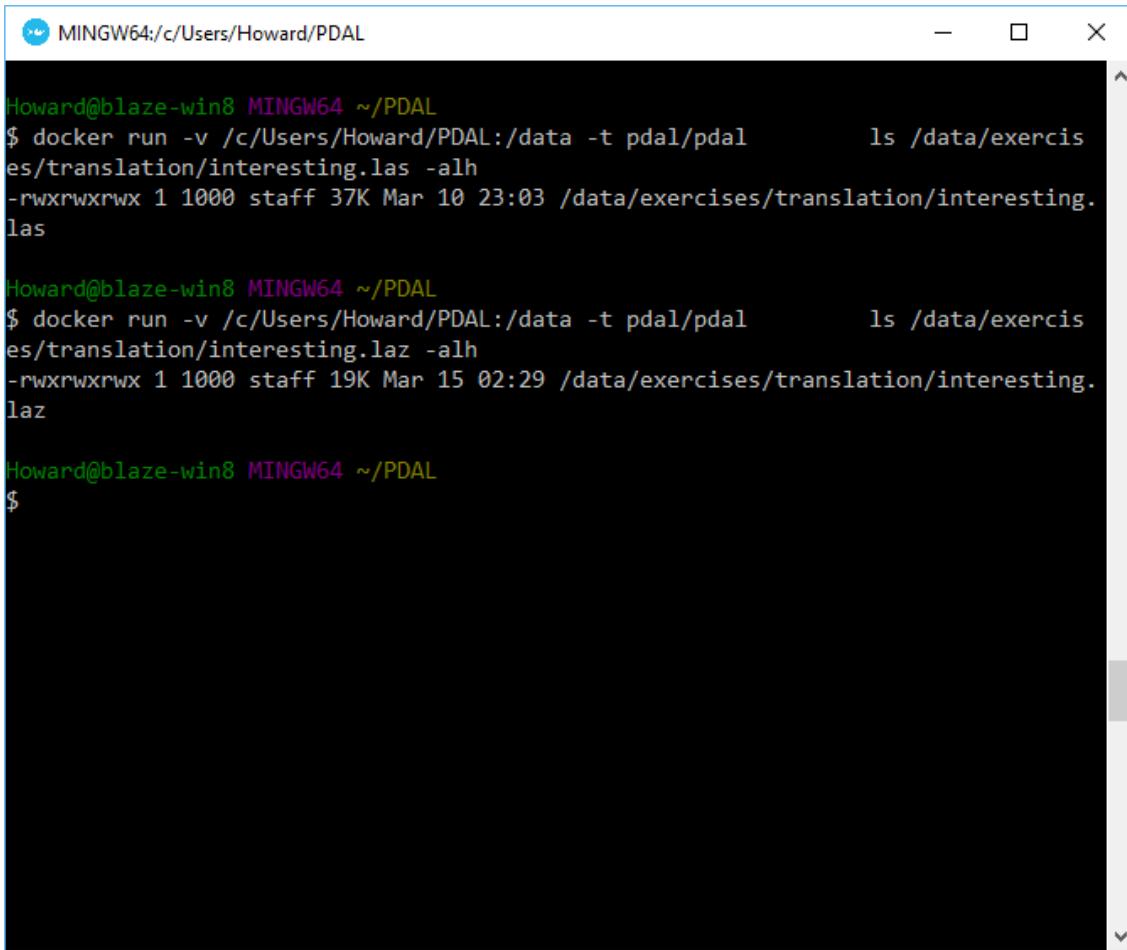
LAS is a very fluffy binary format. Because of the way the data are stored, there is ample redundant information, and **LASzip** (<http://laszip.org>) is an open source solution for compressing this information

1. Verify that the data are in fact compressed:

```
docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
    ls -alh /data/exercises/translation/interesting.laz

docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
    ls -alh /data/exercises/translation/interesting.las
```

PDAL: Point cloud Data Abstraction Library, 1.4.0

A screenshot of a terminal window titled "MINGW64:/c/Users/Howard/PDAL". The window contains three lines of terminal output. The first line shows the command \$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal ls /data/exercises/translation/interesting.las -alh, resulting in a file named interesting.las with permissions -rwxrwxrwx 1 1000 staff 37K Mar 10 23:03. The second line shows the command \$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal ls /data/exercises/translation/interesting.laz -alh, resulting in a file named interesting.laz with permissions -rwxrwxrwx 1 1000 staff 19K Mar 15 02:29. The third line shows the command \$.

Notes

1. Typical [LASzip](http://laszip.org) (<http://laszip.org>) compression is 5:1 to 8:1, depending on the type of [LiDAR](https://en.wikipedia.org/wiki/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) (<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

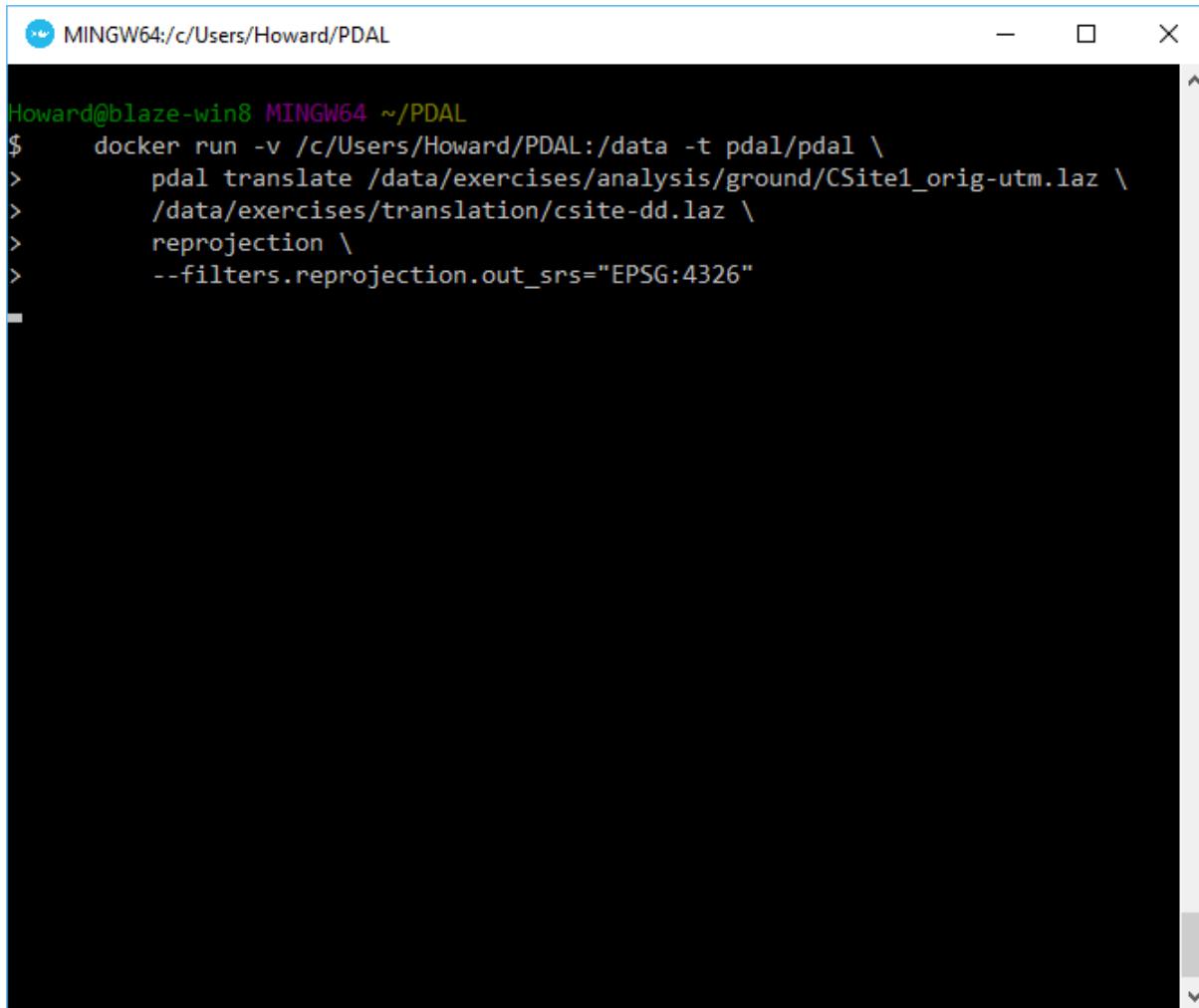
Exercise

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 *Docker Quickstart Terminal*.

```
1 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
2     pdal translate /data/exercises/analysis/ground/CSite1_orig-utm.
3     ↳laz \
4         /data/exercises/translation/csite-dd.laz \
5         reprojection \
5         --filters.reprojection.out_srs="EPSG:4326"
```

PDAL: Point cloud Data Abstraction Library, 1.4.0

A screenshot of a terminal window titled "MINGW64:/c/Users/Howard/PDAL". The command entered is:

```
Howard@blaze-win8 MINGW64 ~/PDAL
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
>     pdal translate /data/exercises/analysis/ground/CSite1_orig-utm.laz \
>     /data/exercises/translation/csite-dd.laz \
>     reprojection \
>     --filters.reprojection.out_srs="EPSG:4326"
```

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

```
docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
    pdal info /data/exercises/translation/csite-dd.laz --all
```

```

],
"statistic": [
  {
    "average": 9.17206603,
    "count": 1366408,
    "maximum": 9.18,
    "minimum": 9.16,
    "name": "X",
    "position": 0
  },
  {
    "average": 48.78746773,
    "count": 1366408,
    "maximum": 48.79,
    "minimum": 48.78,
    "name": "Y",
    "position": 1
  },
  {
    "average": 346.9899008,
    "count": 1366408,
    "maximum": 426.91,
    "minimum": 99.43,
    "name": "Z",
    "position": 2
  },
  {
    "average": 217.015147,
    "count": 1366408
  }
]
}

```

Some formats, like *writers.las* (page 82) 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.

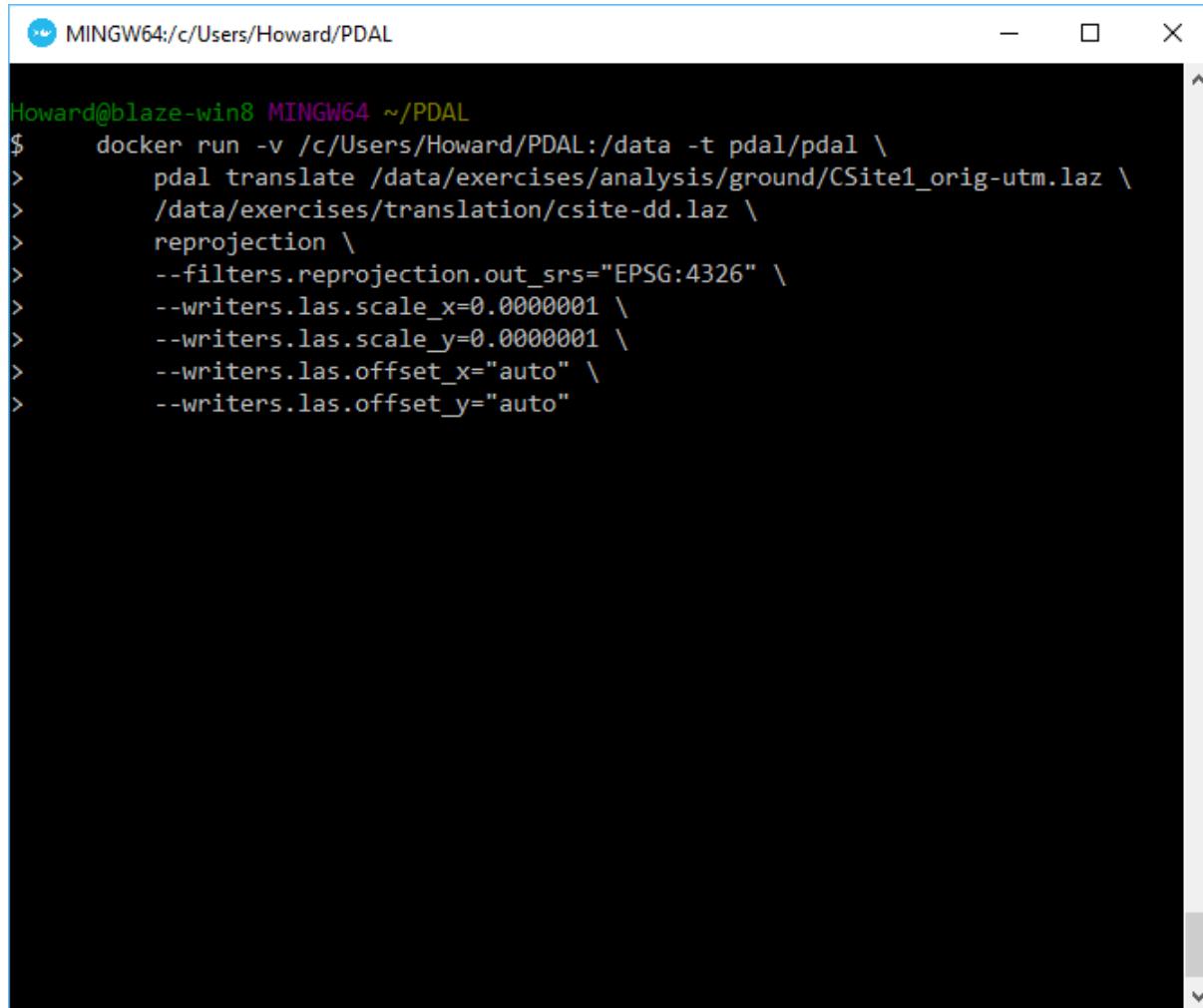
```

1 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
2   pdal translate \
3     /data/exercises/analysis/ground/CSite1_orig-utm.laz \
4     /data/exercises/translation/csite-dd.laz \
5     reprojection \
6     --filters.reprojection.out_srs="EPSG:4326" \

```

PDAL: Point cloud Data Abstraction Library, 1.4.0

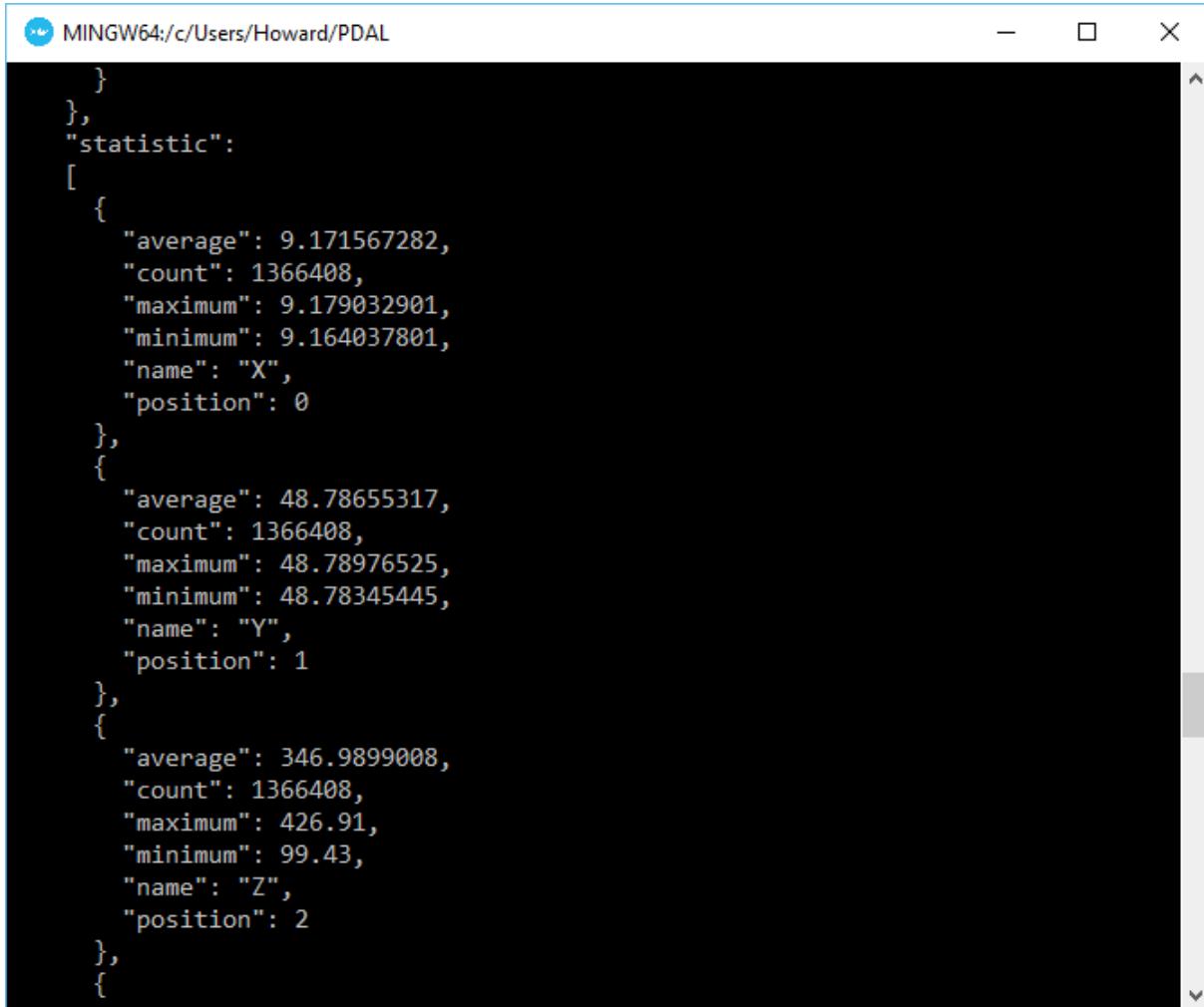
```
7   --writers.las.scale_x=0.0000001 \
8   --writers.las.scale_y=0.0000001 \
9   --writers.las.offset_x="auto" \
10  --writers.las.offset_y="auto"
```



The screenshot shows a terminal window titled "MINGW64:/c/Users/Howard/PDAL". The command entered is:

```
Howard@blaze-win8 MINGW64 ~/PDAL
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
>     pdal translate /data/exercises/analysis/ground/CSite1_orig-utm.laz \
>     /data/exercises/translation/csite-dd.laz \
>     reprojection \
>     --filters.reprojection.out_srs="EPSG:4326" \
>     --writers.las.scale_x=0.0000001 \
>     --writers.las.scale_y=0.0000001 \
>     --writers.las.offset_x="auto" \
>     --writers.las.offset_y="auto"
```

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

A screenshot of a terminal window titled "MINGW64:/c/Users/Howard/PDAL". The window contains JSON output from a PDAL command. The output shows statistics for three dimensions: X, Y, and Z. For X, the average is 9.171567282, count is 1366408, maximum is 9.179032901, minimum is 9.164037801, name is "X", and position is 0. For Y, the average is 48.78655317, count is 1366408, maximum is 48.78976525, minimum is 48.78345445, name is "Y", and position is 1. For Z, the average is 346.9899008, count is 1366408, maximum is 426.91, minimum is 99.43, name is "Z", and position is 2.

```
{
  },
  "statistic": [
    {
      "average": 9.171567282,
      "count": 1366408,
      "maximum": 9.179032901,
      "minimum": 9.164037801,
      "name": "X",
      "position": 0
    },
    {
      "average": 48.78655317,
      "count": 1366408,
      "maximum": 48.78976525,
      "minimum": 48.78345445,
      "name": "Y",
      "position": 1
    },
    {
      "average": 346.9899008,
      "count": 1366408,
      "maximum": 426.91,
      "minimum": 99.43,
      "name": "Z",
      "position": 2
    }
  ]
}
```

Notes

1. *filters.reprojection* (page 152) 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.

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](#) (page 286) to visualize 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 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
2     pdal info \
3     /data/exercises/analysis/density/uncompahgre.laz \
4     --boundary
5
```

```

MINGW64:/c/Users/Howard/PDAL
Howard@blaze-win8 MINGW64 ~/PDAL
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal pdal info /data/exercises/analysis/density/uncompahgre.laz --boundary
{
  "boundary": {
    "area": 90179889.42,
    "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 4208999.71639143, 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 4213725.27140854, 247991.21607113 4213799.10820569, 248119.10515522 4214168.29219140, 247735.43790296 4214242.12898854, 247905.95668174 4214537.47617711, 247479.65973478 4215275.84414853, 247522.28942948 4215497.35453996, 247223.88156661 4215718.86493139, 247394.40034539 4215866.53852567, 247095.99248252 4216235.72251139, 247138.62217722 4216457.23290281, 246840.21431435 4216974.09048281, 246882.84400904 4217195.60087424, 246712.32523026 4217343.27446852, 246754.95492496 4217712.45845423, 246584.43614617 4217860.13204852, 24754.95492496 4218007.80564280, 246499.17675678 4218007.80564280, 246627.06584087

```

... 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](#) (page 286). 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 123). We can leverage this capability to output a contiguous boundary of the `uncompahgre.laz` file.

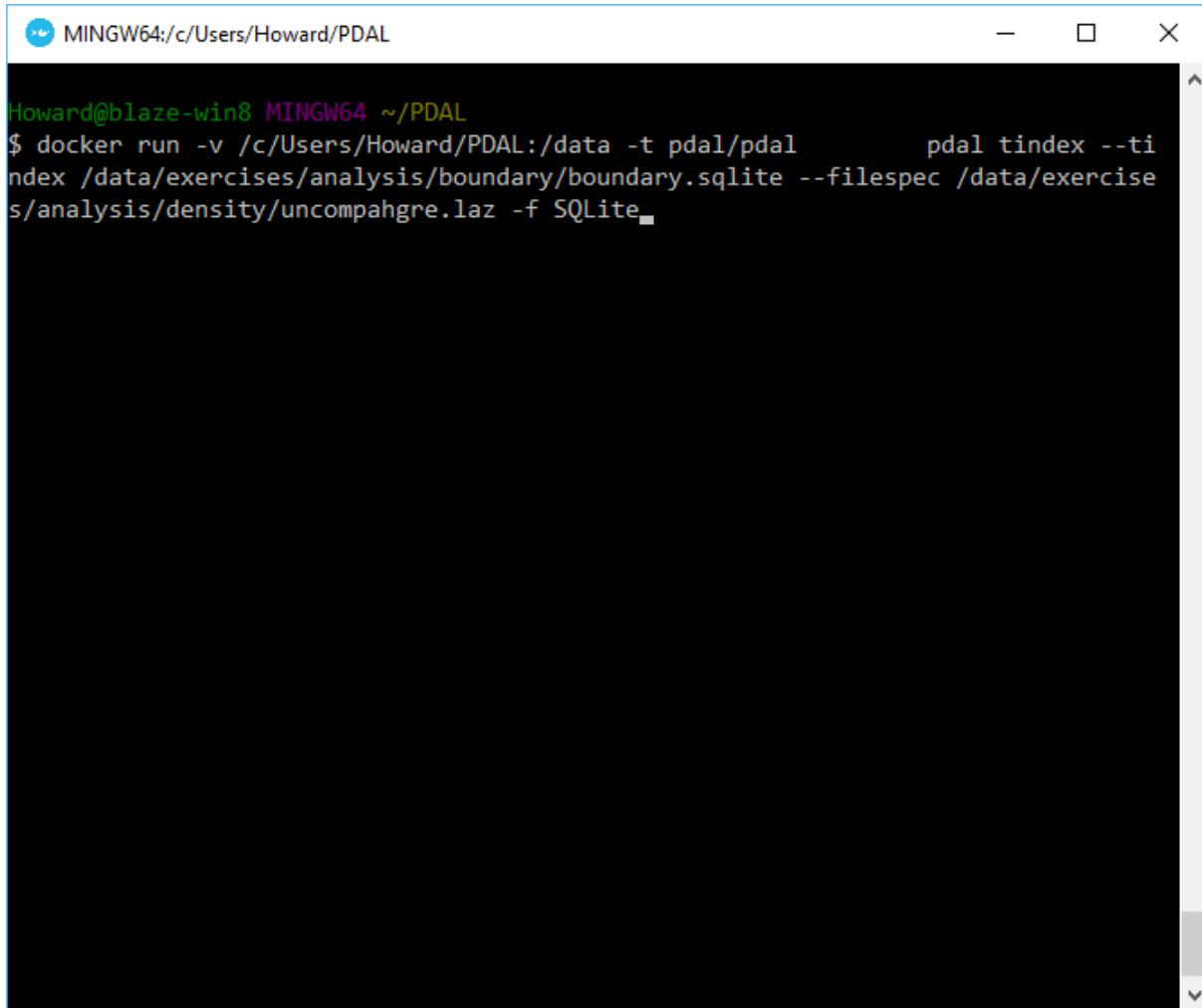
```

1 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
2   pdal tindex \
3     --tindex /data/exercises/analysis/boundary/boundary.sqlite \
4     --filespec /data/exercises/analysis/density/uncompahgre.laz \

```

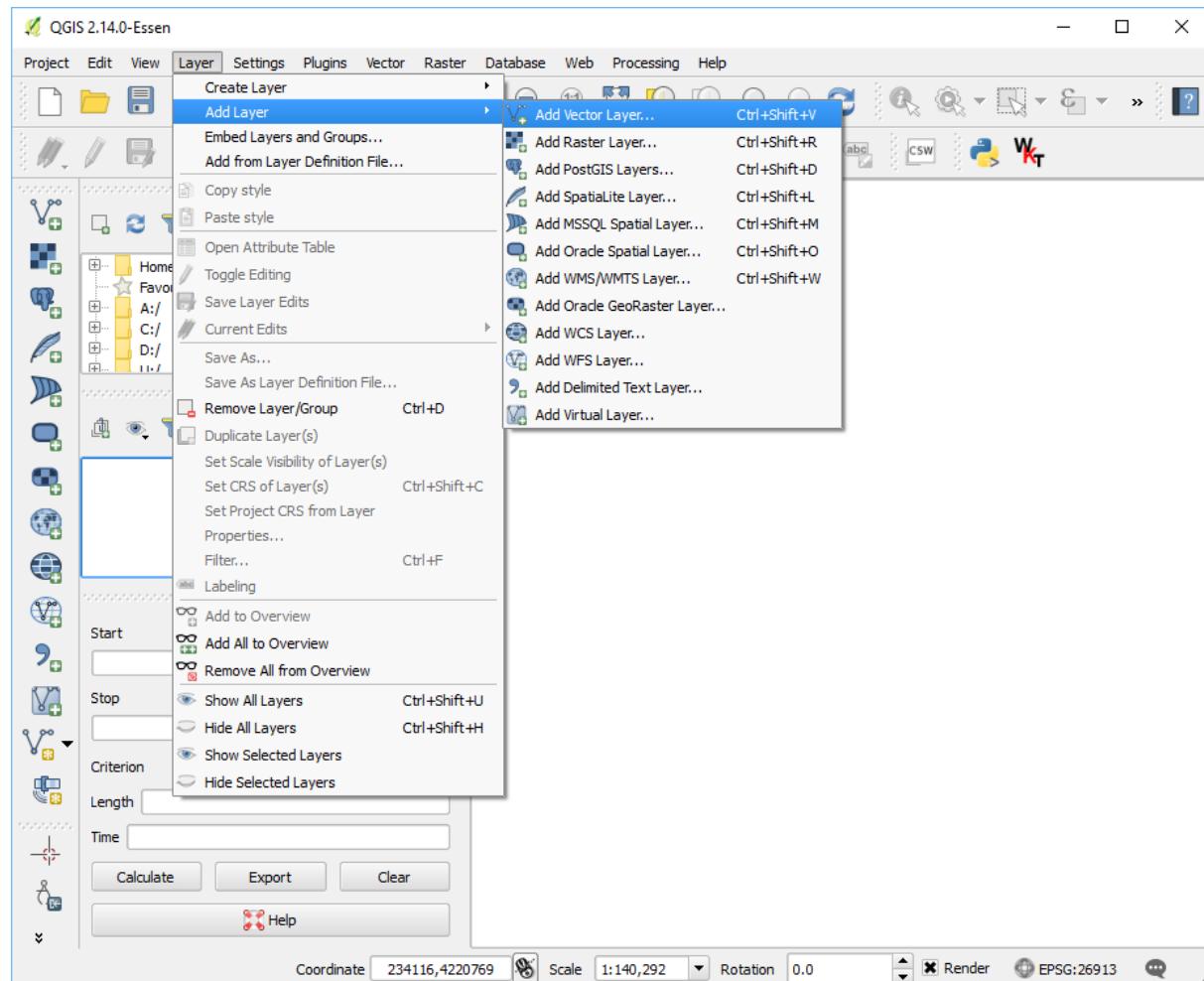
PDAL: Point cloud Data Abstraction Library, 1.4.0

5 -f SQLite

A screenshot of a terminal window titled "MINGW64:/c/Users/Howard/PDAL". The window shows a command being run in a Docker container. The command is: \$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal pdal tindex --tile-index /data/exercises/analysis/boundary/boundary.sqlite --filespec /data/exercises/analysis/density/uncompahgre.laz -f SQLite. The terminal window has a light gray header bar and a dark gray body. The command is visible in the body of the window.

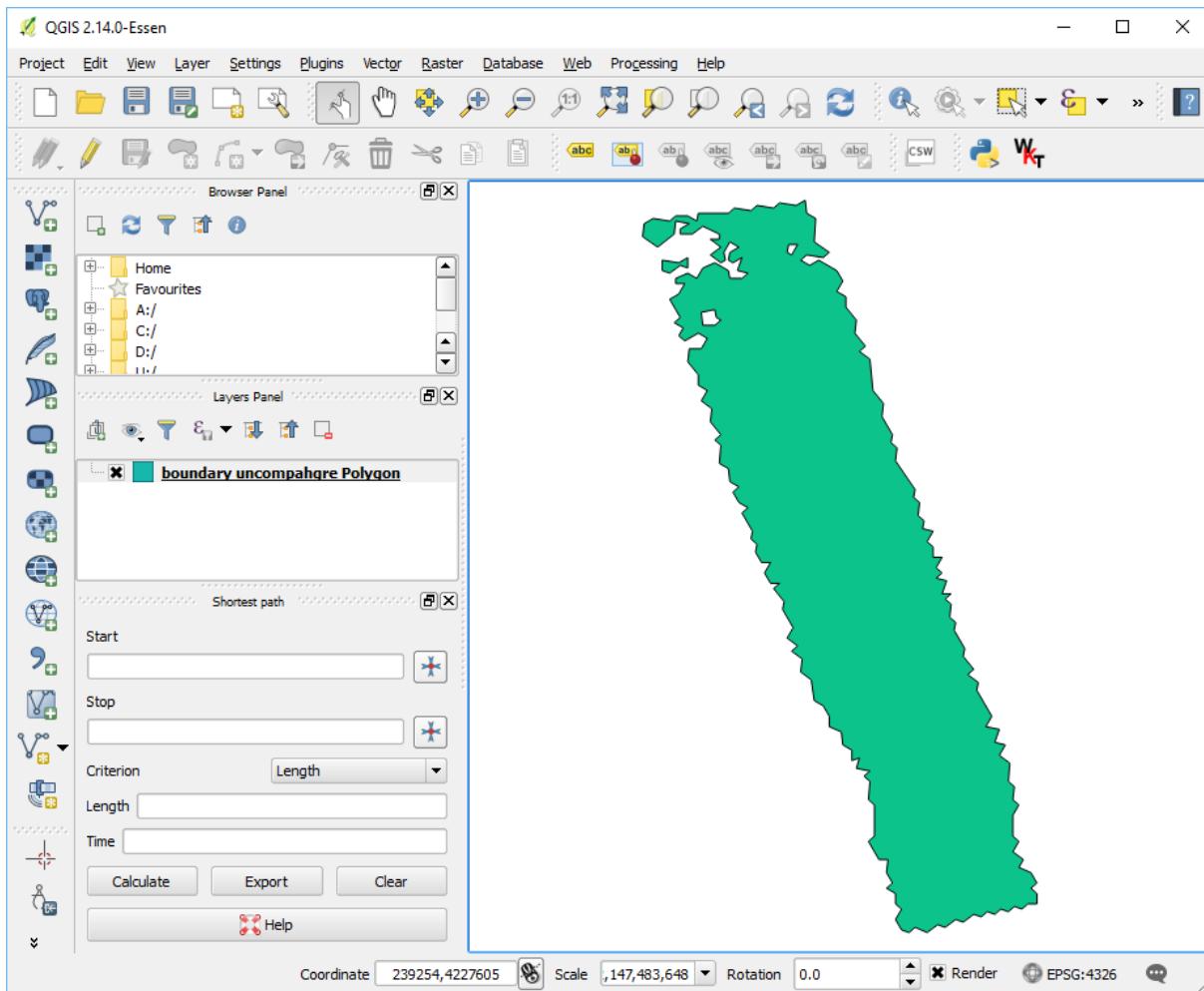
Once we've run the *tindex* (page 29), we can now visualize our output:

Open *QGIS* (page 286) and select *Add Vector Layer*:



Navigate to the exercises/analysis/boundary directory and then open the boundary.sqlite file:

PDAL: Point cloud Data Abstraction Library, 1.4.0



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 123) can also be used by the `density` (page 19) to generate a tessellated surface. See the *Visualizing acquisition density* (page 328) example for steps to achieve this.
3. The `tindex` (page 29) 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 `t index` application. You can also use this command to merge data together (query across boundaries, for example).

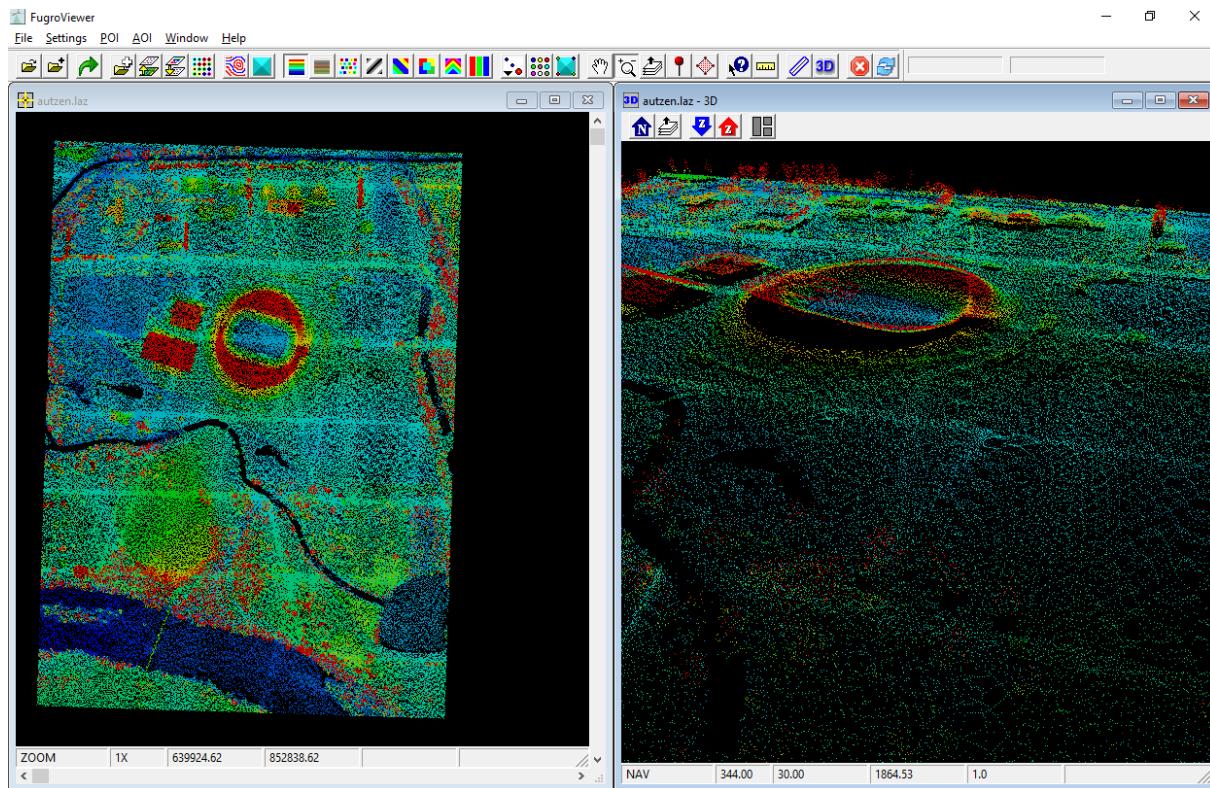
Clipping data with polygons

This exercise uses PDAL to apply to clip data with polygon geometries.

Note: This exercise is an adaption of the *PDAL tutorial* (page 206).

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.



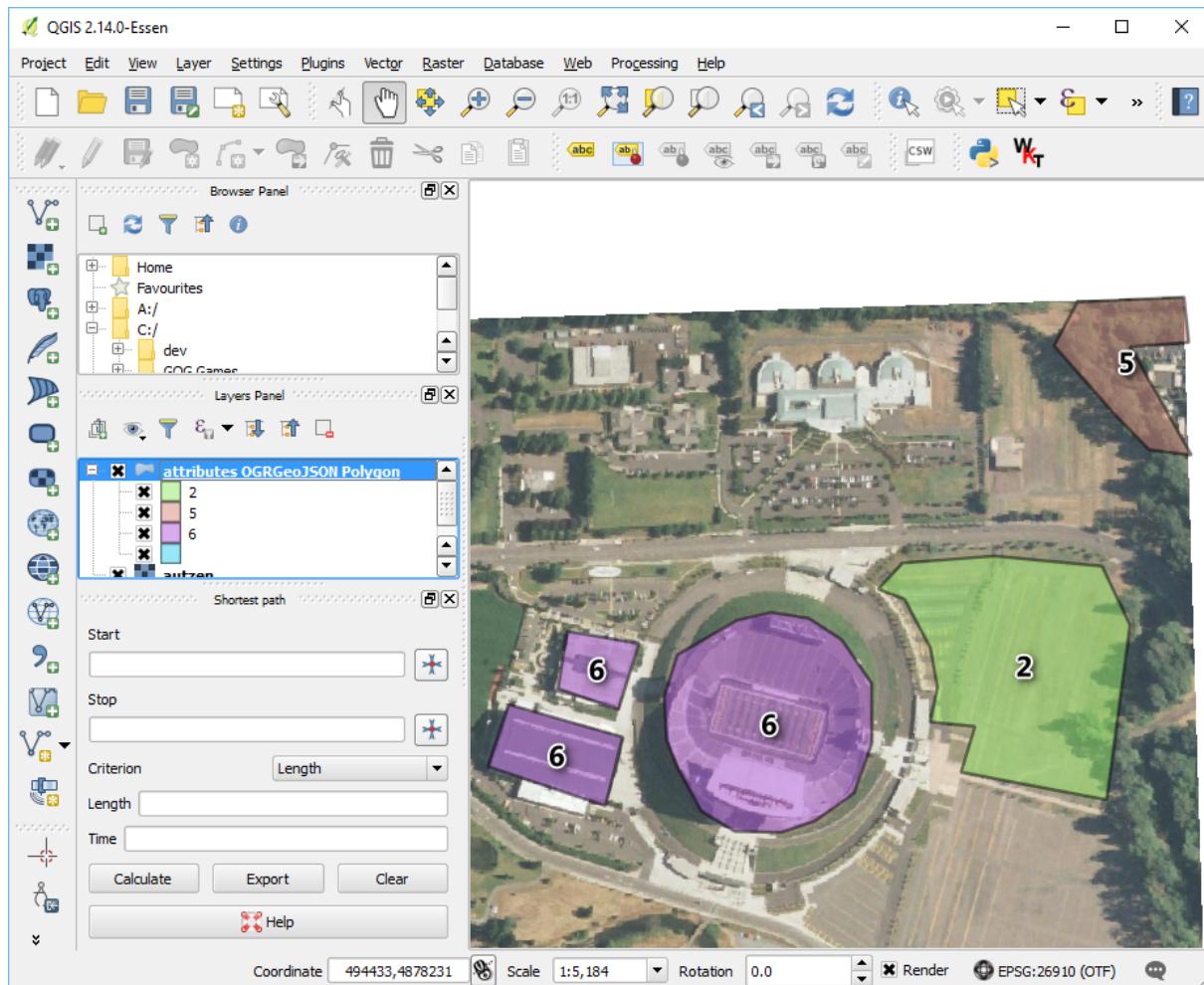
Data preparation

The data are mixed in two different coordinate systems. The [LAZ](#) (page 57) 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) (http://www.gdal.org/drv_vrt.html) for more details.

Pipeline breakdown

```
{  
  "pipeline": [  
    "/data/exercises/analysis/clipping/autzen.laz",  
    {  
      "column": "CLS",  
      "datasource": "/data/exercises/analysis/clipping/  
      ↵attributes.vrt",  
      "dimension": "Classification",  
      "layer": "OGRGeoJSON",  
      "type": "filters.attribute"  
    },  
    {  
      "limits": "Classification[6:6]",  
      "type": "filters.range"  
    },  
    "/data/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.attribute

The `filters.attribute` (page 103) filter allows you to assign values for coincident polygons. Using the VRT we defined in *Data preparation* (page 312), `filters.attribute` (page 103) 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 151) to keep points with Classification values in the range of 6:6.

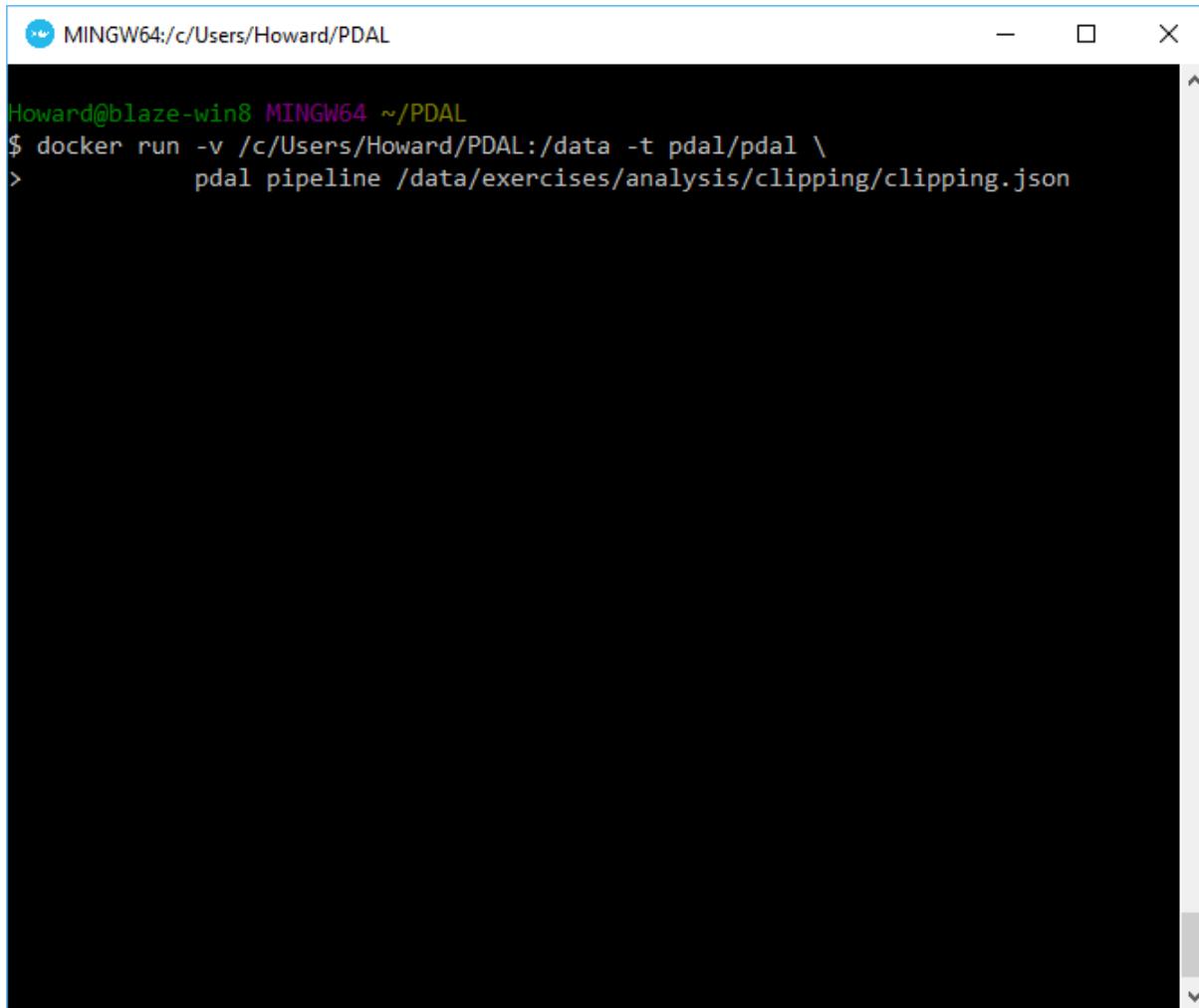
3. Writer

- We will write our content back out using an `writers.las` (page 82).

Execution

Invoke the following command, substituting accordingly, in your *Docker Quickstart Terminal*:

```
1 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
2     pdal pipeline \
3     /data/exercises/analysis/clipping/clipping.json
```

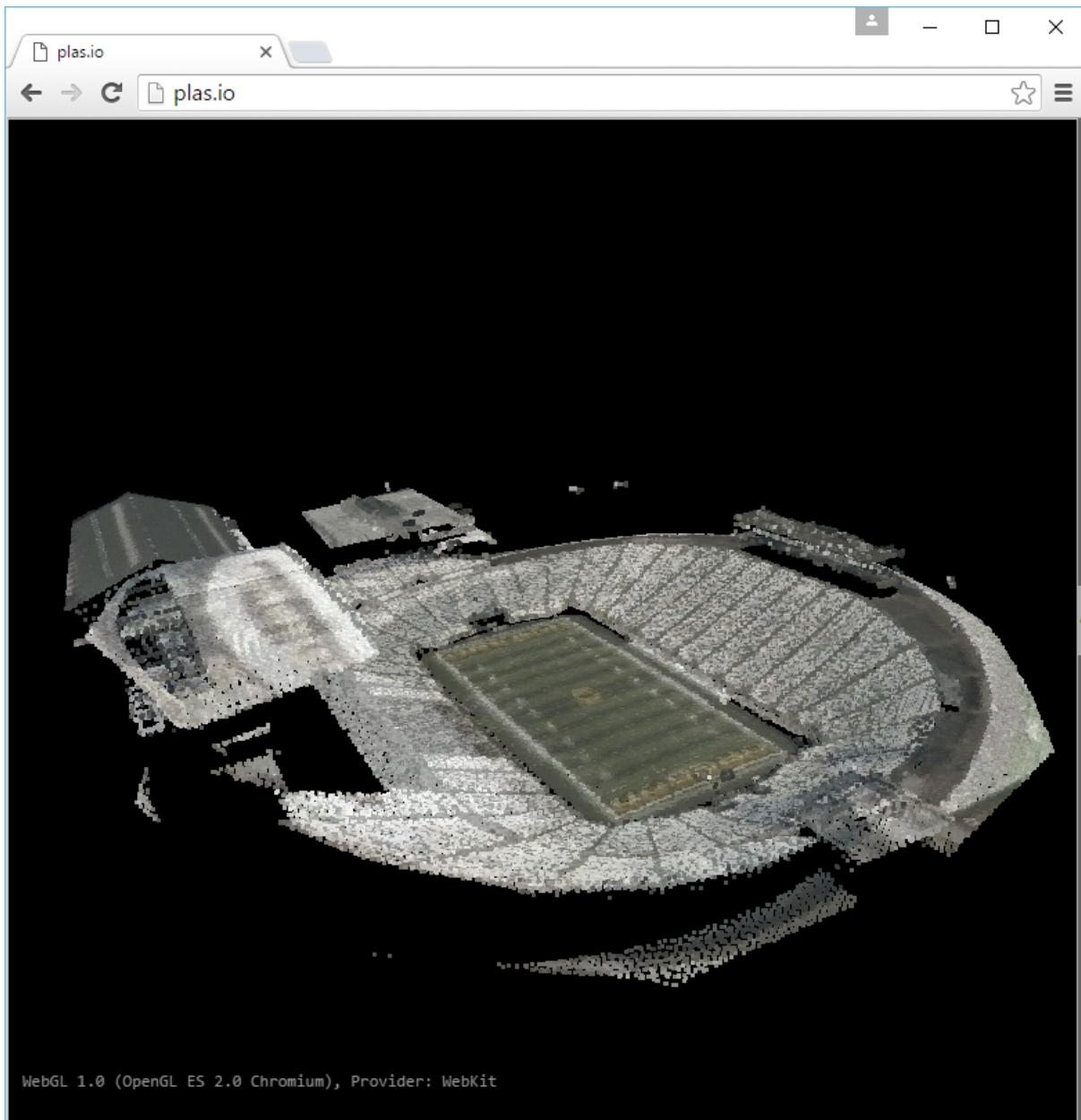


The screenshot shows a terminal window titled "MINGW64:/c/Users/Howard/PDAL". The command entered is:

```
Howard@blaze-win8 MINGW64 ~/PDAL
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
>     pdal pipeline /data/exercises/analysis/clipping/clipping.json
```

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 simply opened the file using the <http://plas.io> website.



Notes

1. *filters.attribute* (page 103) 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 101) 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          "/data/exercises/analysis/colorization/uncompahgre.laz",
4          {
5              "type": "filters.colorization",
6              "raster": "/data/exercises/analysis/colorization/casi-
7              ↪2015-04-29-weekly-mosaic.tif"
8          },
9          {
10             "type": "filters.range",
11             "limits": "Red[1:]"
12         },
13         {
14             "type": "writers.las",
15             "compression": "true",
16             "minor_version": "2",
17             "dataformat_id": "3",
18             "filename": "/data/exercises/analysis/colorization/
19             ↪uncompahgre-colored.laz"
20         }
21     ]
22 }
```

```
20 }  
21 }
```

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.

```
"/data/exercises/analysis/colorization/uncompahgre.laz",
```

2. filters.colorization

The [filters.colorization](#) (page 111) 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": "/data/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 151) 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": "/data/exercises/colorization/analysis/uncompahgre-  
    ↵colored.laz"  
}
```

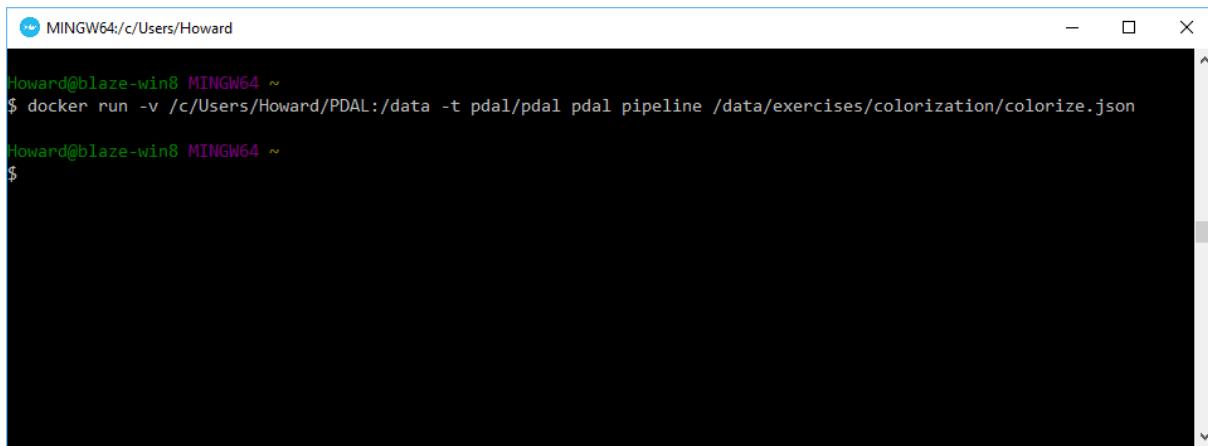
1. compression: [LASzip](http://laszip.org) (<http://laszip.org>) data is ~6x smaller than ASPRS LAS.
2. minor_version: We want to make sure to output LAS 1.2, which will provide the widest compatibility with other softwares that can consume LAS.
3. dataformat_id: Format 3 supports both time and color information

Note: [writers.las](#) (page 82) 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 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \  
2     pdal pipeline \  
3     /data/exercises/analysis/colorization/colorize.json
```

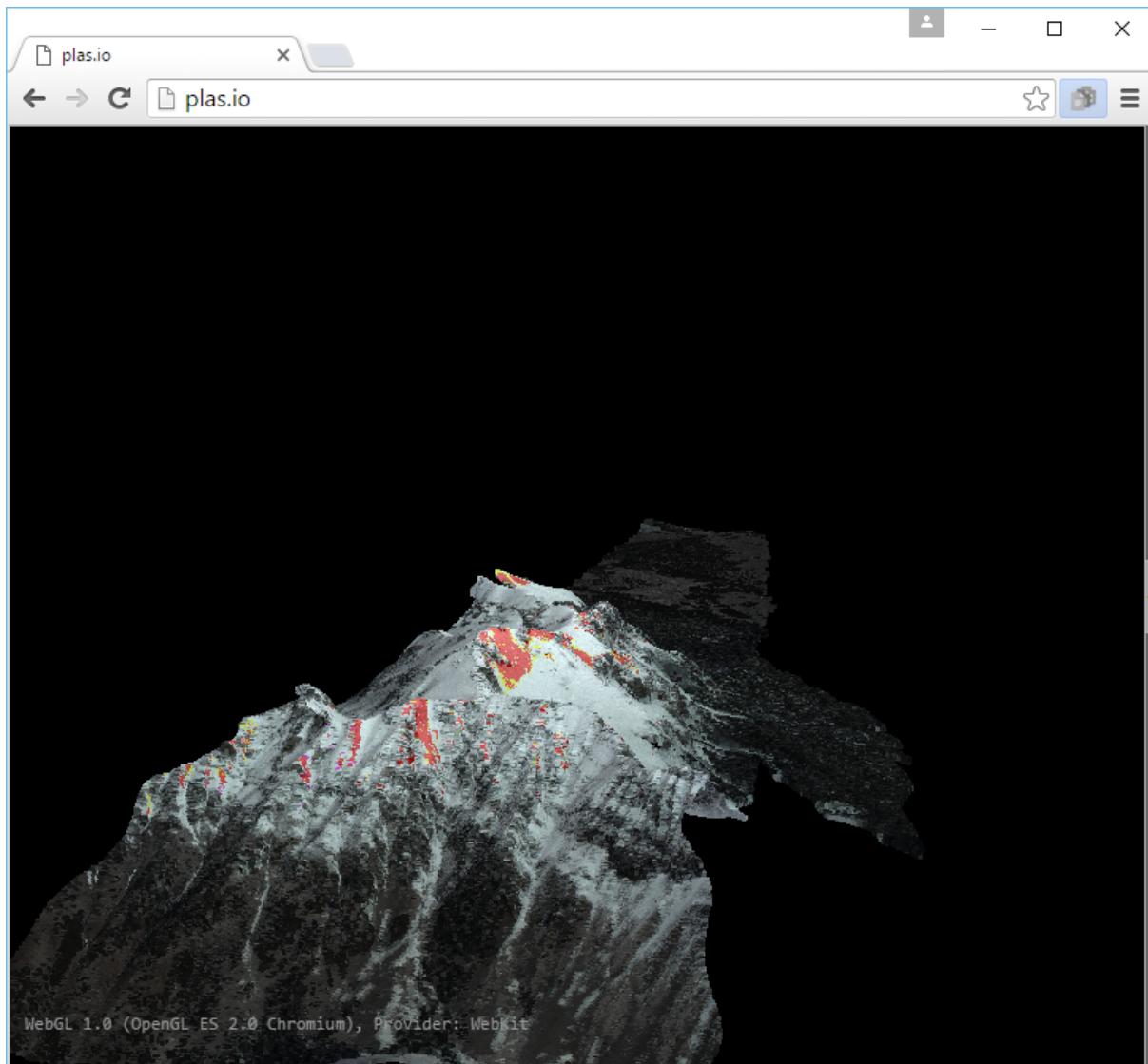


A screenshot of a terminal window titled "MINGW64:c/Users/Howard". The window shows a command being run in a Docker container:

```
Howard@blaze-win8 MINGW64 ~
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal pdal pipeline /data/exercises/colorization/colorize.json
Howard@blaze-win8 MINGW64 ~
$
```

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 111) 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 ALS collection.

Warning: Our default *Install Docker* (page 9) machine instance is probably going to run out of memory for this operation (it only has 1gb). We may need to recreate it with the following commands to increase the available memory:

1. Remove the existing machine instances

```
$ docker-machine rm default
```

2. Create a new one with 2gb of memory

```
$ docker-machine rm default
```

Exercise

PDAL provides a *filter* (page 101) through **PCL** (<http://pointclouds.org>) to apply a statistical filter to data.

Because this operation is somewhat complex, we are going to use a pipeline to define it.

```
{
  "pipeline": [
    "/data/exercises/analysis/denoising/18TWK820985.laz",
    {
      "type": "filters.outlier",
      "method": "statistical",
      "extract": "true",
      "multiplier": 3,
      "mean_k": 8
    }
  ]
}
```

```
        },
        {
            "type": "filters.range",
            "limits": "Z[-100:3000]"
        },
        {
            "type": "writers.las",
            "compression": "true",
            "minor_version": "2",
            "dataformat_id": "0",
            "filename": "/data/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.

```
"/data/exercises/analysis/denoising/18TWK820985.laz",
```

2. filters.outlier

The `filters.outlier` (page 137) PDAL filter does most of the work for this operation.

```
{
    "type": "filters.outlier",
    "method": "statistical",
```

```

    "extract": "true",
    "multiplier": 3,
    "mean_k": 8
},

```

3. filters.range

Even with the [filters.outlier](#) (page 137) 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 are going to use [:filters.range](#) (page 151) to keep only points that are within the range $-100 \leq Z \leq 3000$.

```

{
  "type": "filters.range",
  "limits": "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": "/data/exercises/analysis/denoising/clean.laz"
}

```

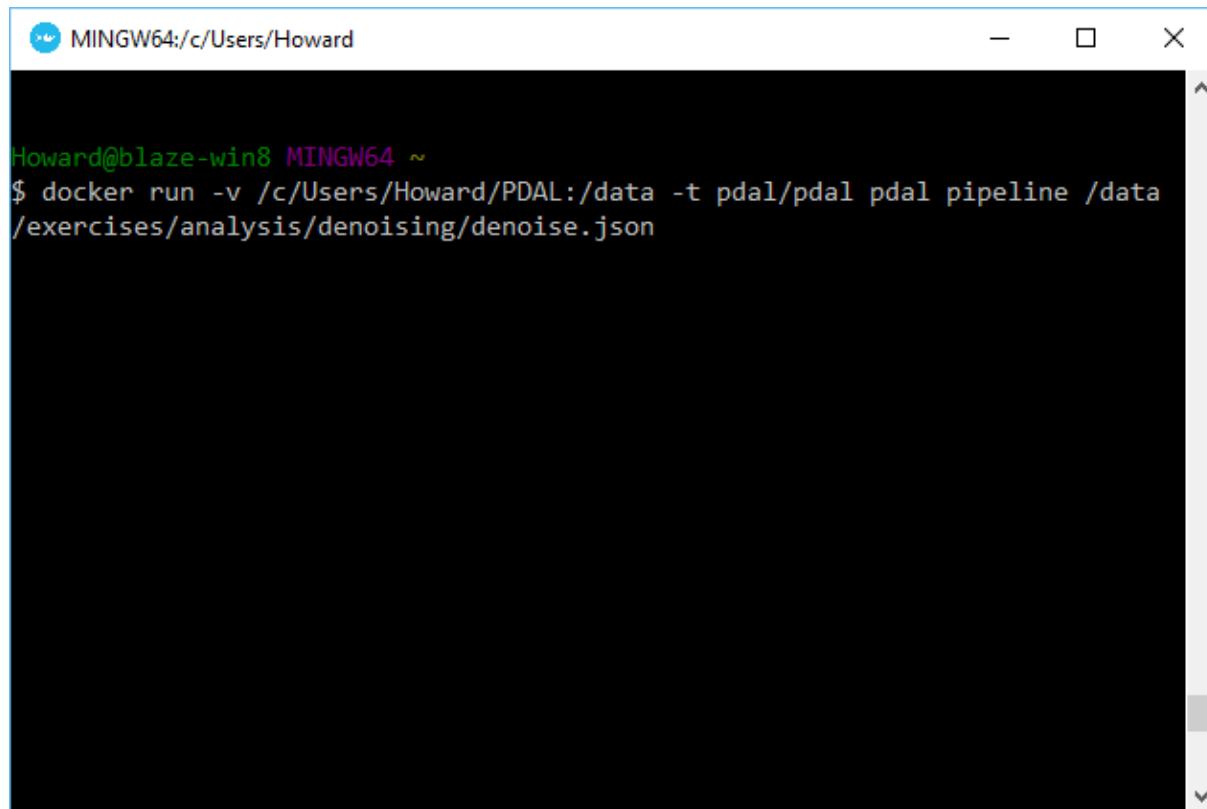
1. compression: [LASzip](#) (<http://laszip.org>) data is ~6x smaller than ASPRS LAS.
2. minor_version: We want to make sure to output LAS 1.2, which will provide the widest compatibility with other softwares that can consume LAS.
3. dataformat_id: Format 3 supports both time and color information

Note: [writers.las](#) (page 82) 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*:

```
docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
    pdal pipeline \
    /data/exercises/analysis/denoising/denoise.json
```

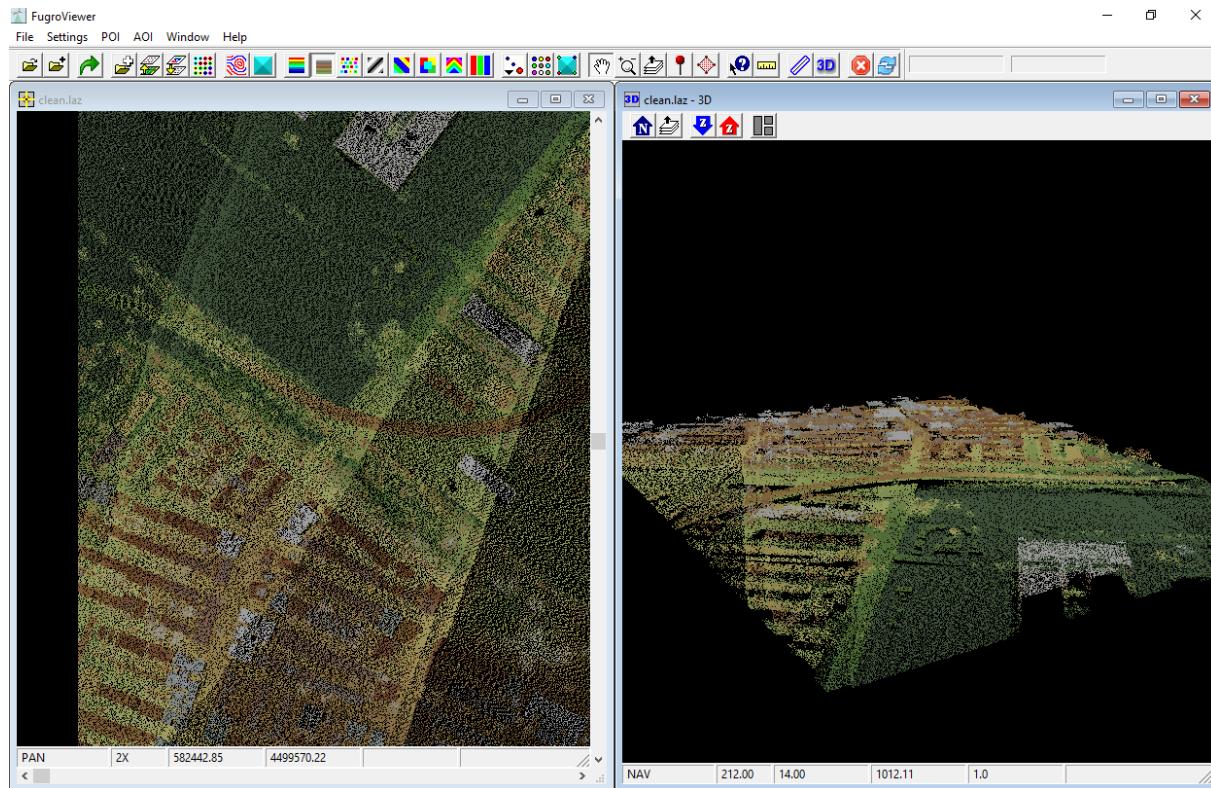


The screenshot shows a terminal window titled "MINGW64:/c/Users/Howard". The command entered is:

```
Howard@blaze-win8 MINGW64 ~
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal pdal pipeline /data
/exercises/analysis/denoising/denoise.json
```

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 137) requires the entire set in memory to process. If you have really large files, you are going to need to `split` (page 157) 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.

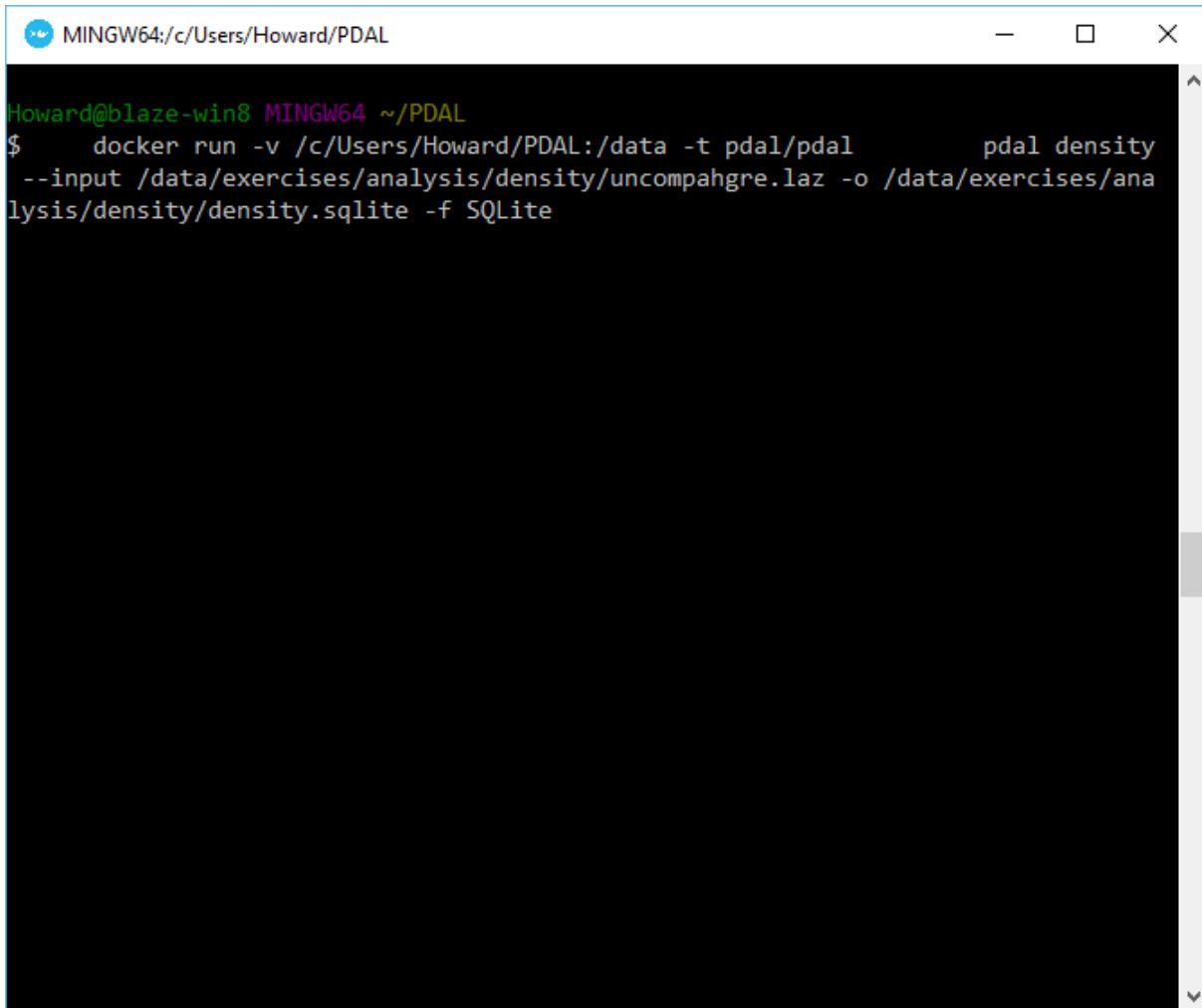
Exercise

PDAL provides an *application* (page 19) to compute a vector field of hexagons computed with *filters.hexbin* (page 123). 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 *Docker Quickstart Terminal*:

```
1 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
2     pdal density \
3     /data/exercises/analysis/density/uncompahgre.laz \
4     -o /data/exercises/analysis/density/density.sqlite \
5     -f SQLite
```



The screenshot shows a terminal window titled "MINGW64:/c/Users/Howard/PDAL". The command entered is:

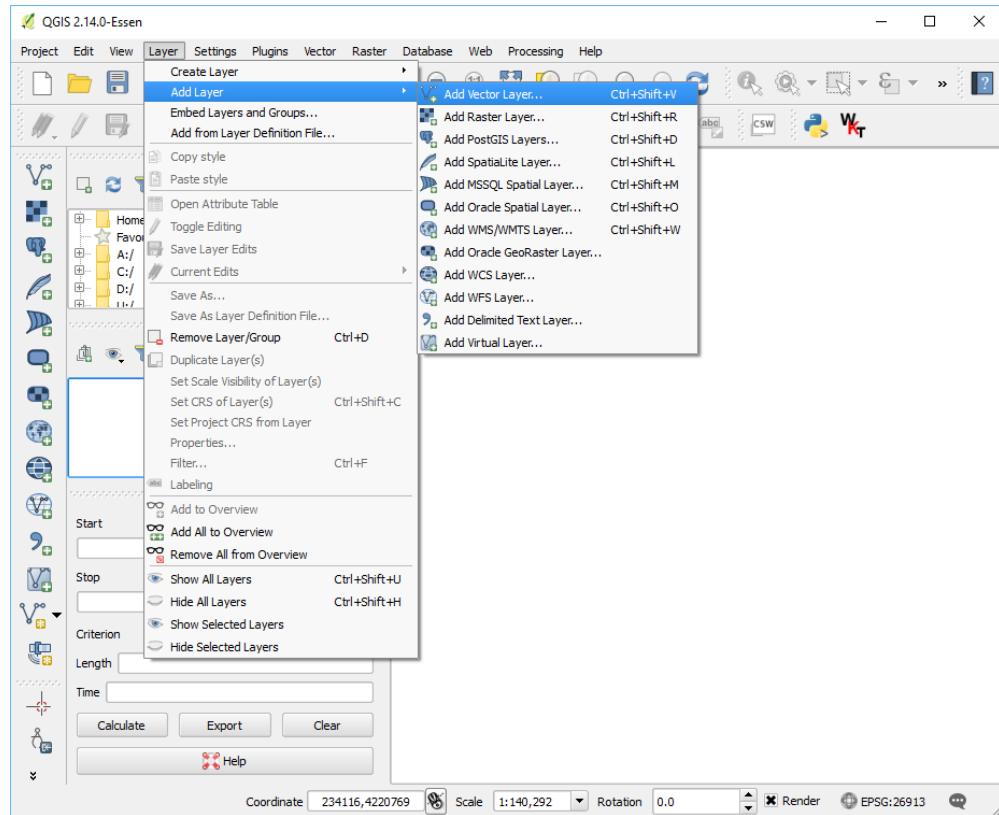
```
Howard@blaze-win8 MINGW64 ~/PDAL
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal      pdal density
--input /data/exercises/analysis/density/uncompahgre.laz -o /data/exercises/ana
lysis/density/density.sqlite -f SQLite
```

Visualization

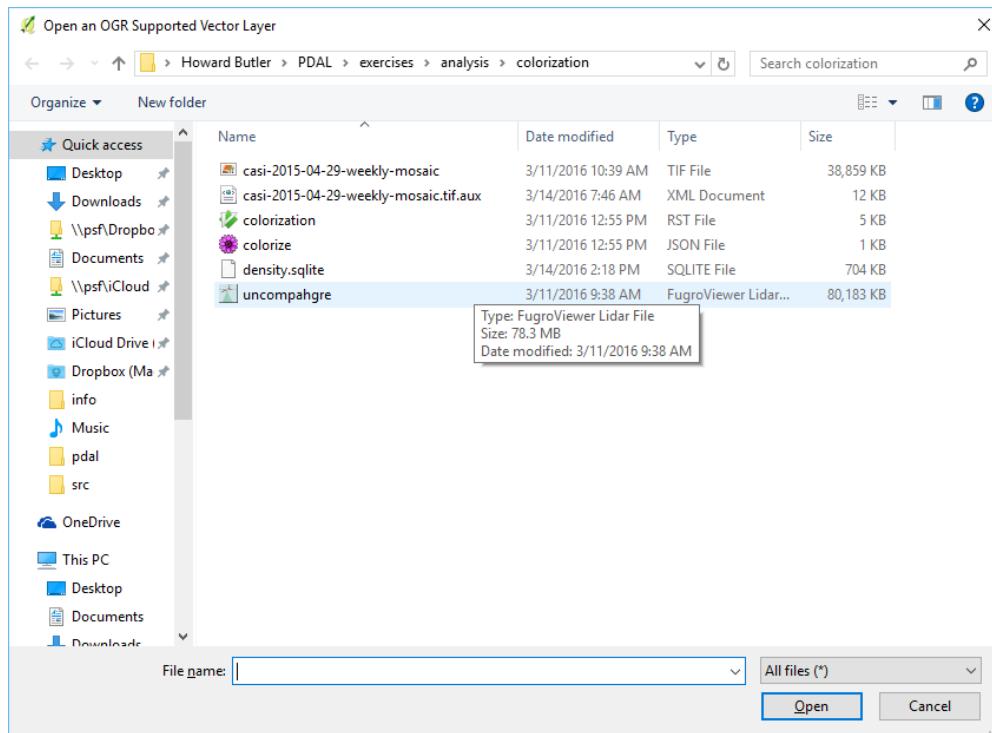
The command uses GDAL to output a [SQLite](http://sqlite.org) (<http://sqlite.org>) file containing hexagon polygons. We will now use [QGIS](http://qgis.org) (<http://qgis.org>) to visualize them.

1. Add a vector layer

PDAL: Point cloud Data Abstraction Library, 1.4.0

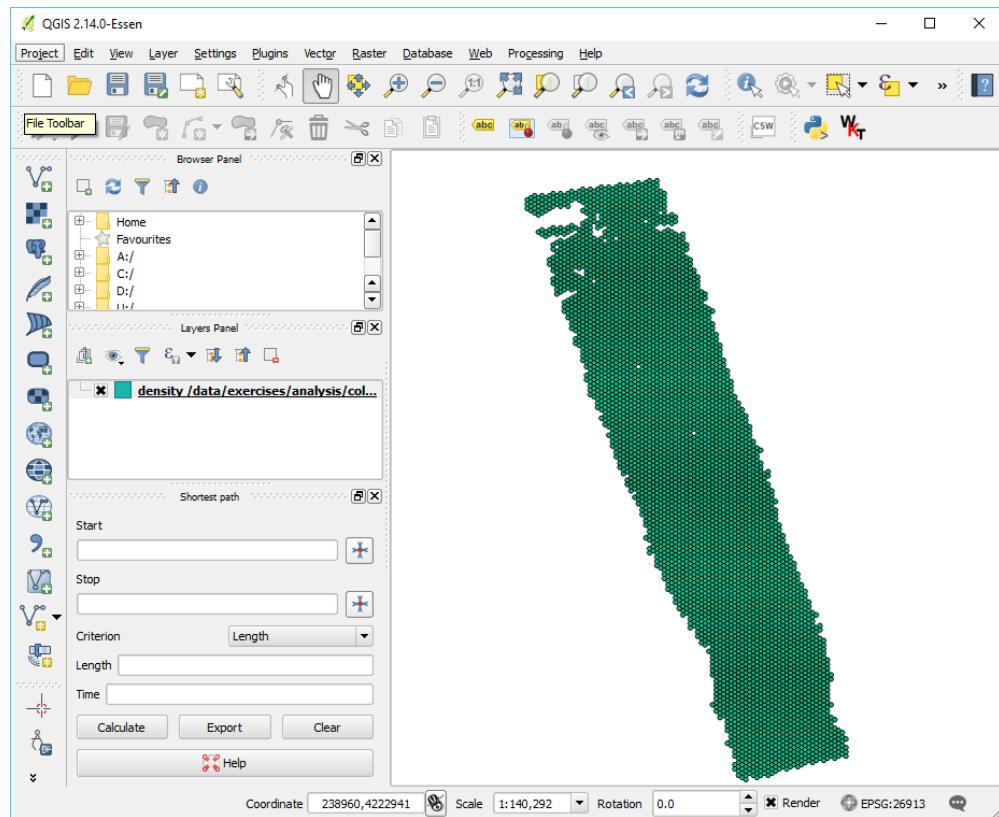


2. Navigate to the output directory

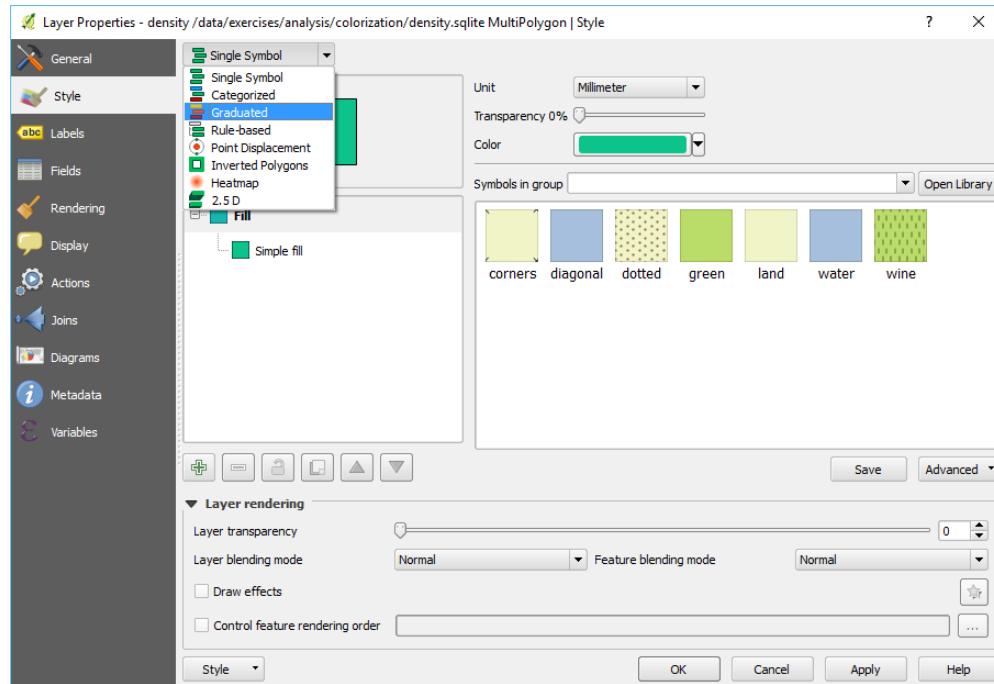


3. Add the `density.sqlite` file to the view

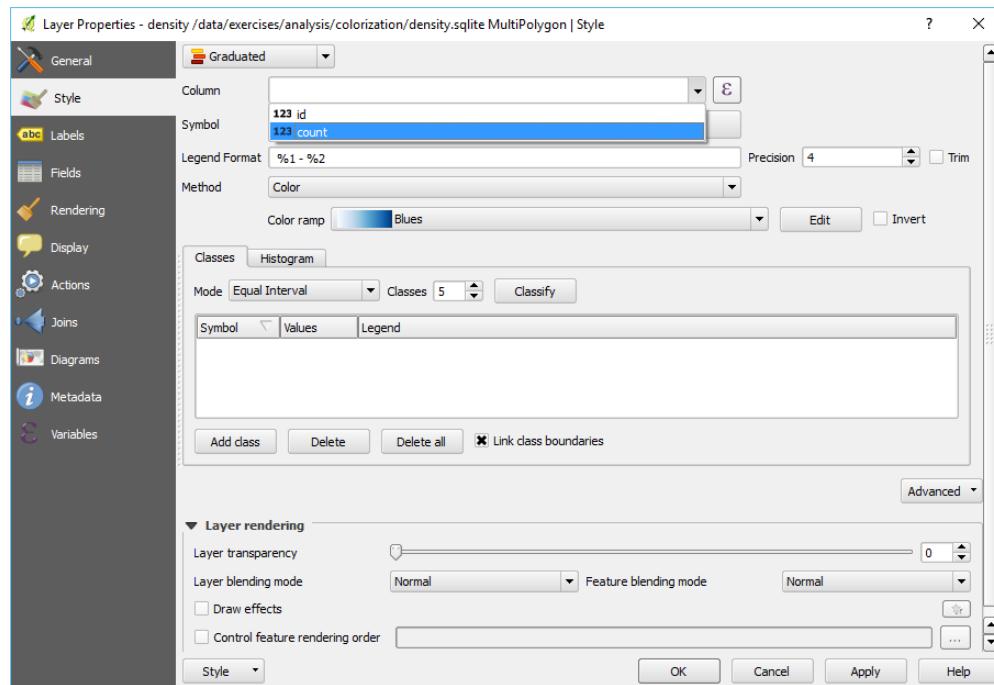
PDAL: Point cloud Data Abstraction Library, 1.4.0



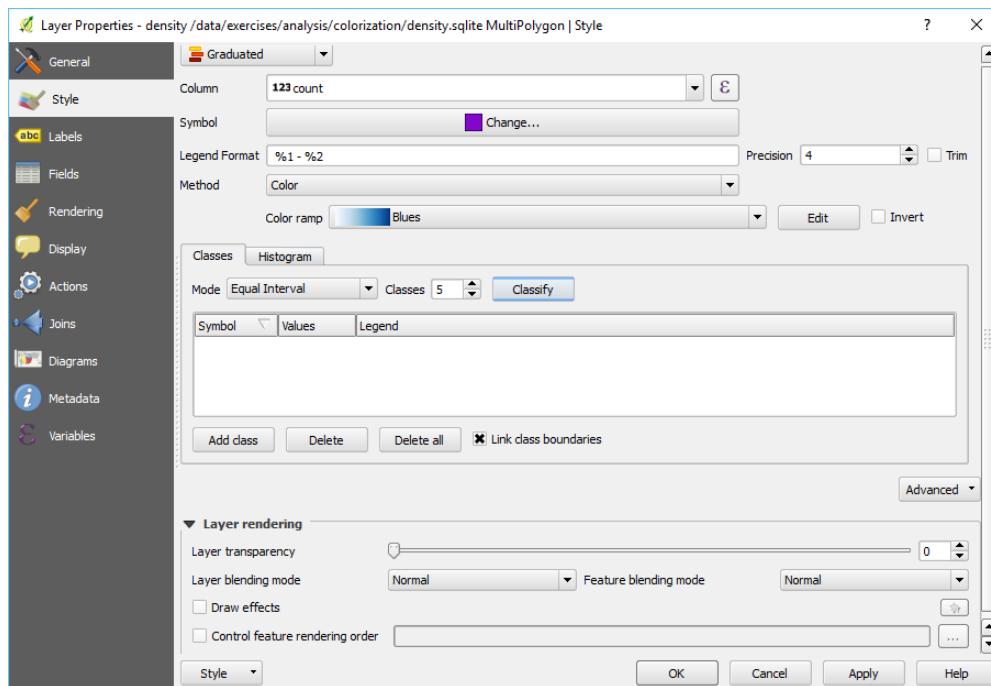
4. Right click on the *density.sqlite* layer in the *Layers* panel and then choose Properties.
5. Pick the Graduated drop down



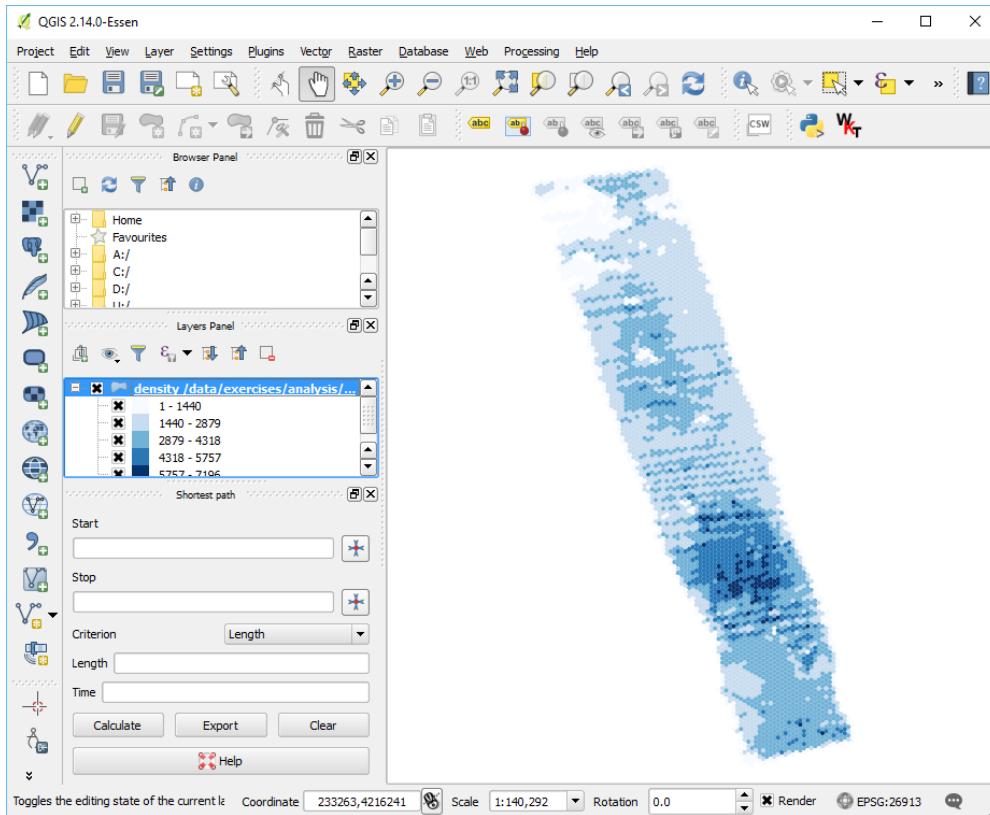
6. Choose the Count column to visualize



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 123).

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 (<http://pdal.io/>)'s [tindex](#) (page 29).

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 excrise is an adaptation of the [Performing Poisson Sampling of Point Clouds Using Dart Throwing](#) (page 219) 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 328) 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 154)
- Random using a combination of [filters.decimation](#) (page 116) and [filters.randomize](#) (page 150)
- Voxel using [filters.voxelgrid](#) (page 160)

In this exercise, we are going to thin with the Poisson method, but the concept should operate similarly for the [filters.voxelgrid](#) (page 160) approach too. See [Performing Poisson Sampling of Point Clouds Using Dart Throwing](#) (page 219) for description of how to randomly filter.

Command

Invoke the following command, substituting accordingly, in your *Docker Quickstart Terminal*:

```
1 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
2     pdal translate \
3     /data/exercises/analysis/density/uncompahgre.laz \
4     /data/exercises/analysis/thinning/uncompahgre-thin.laz \
5     sample \
6     --filters.sample.radius=20
```

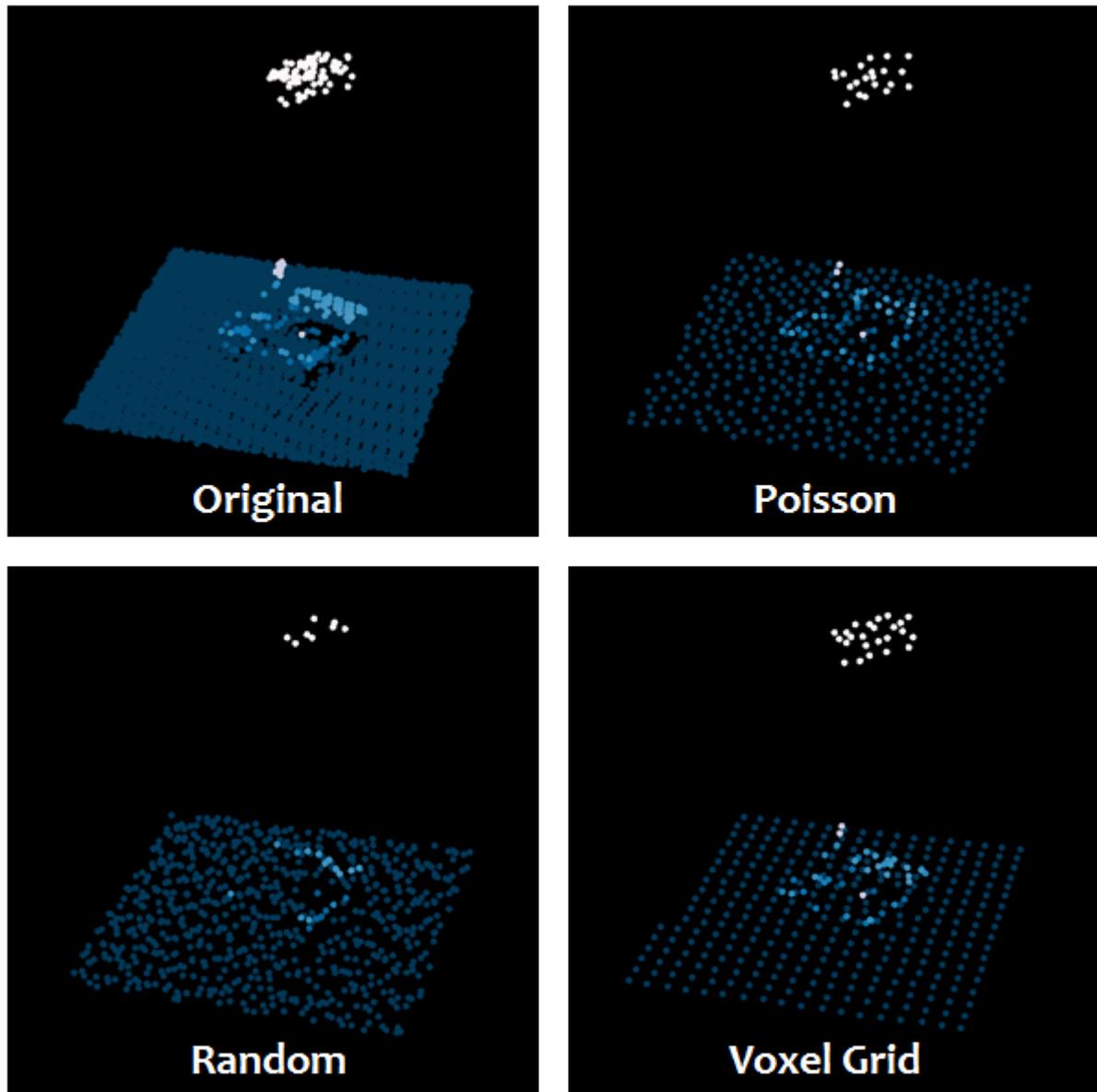
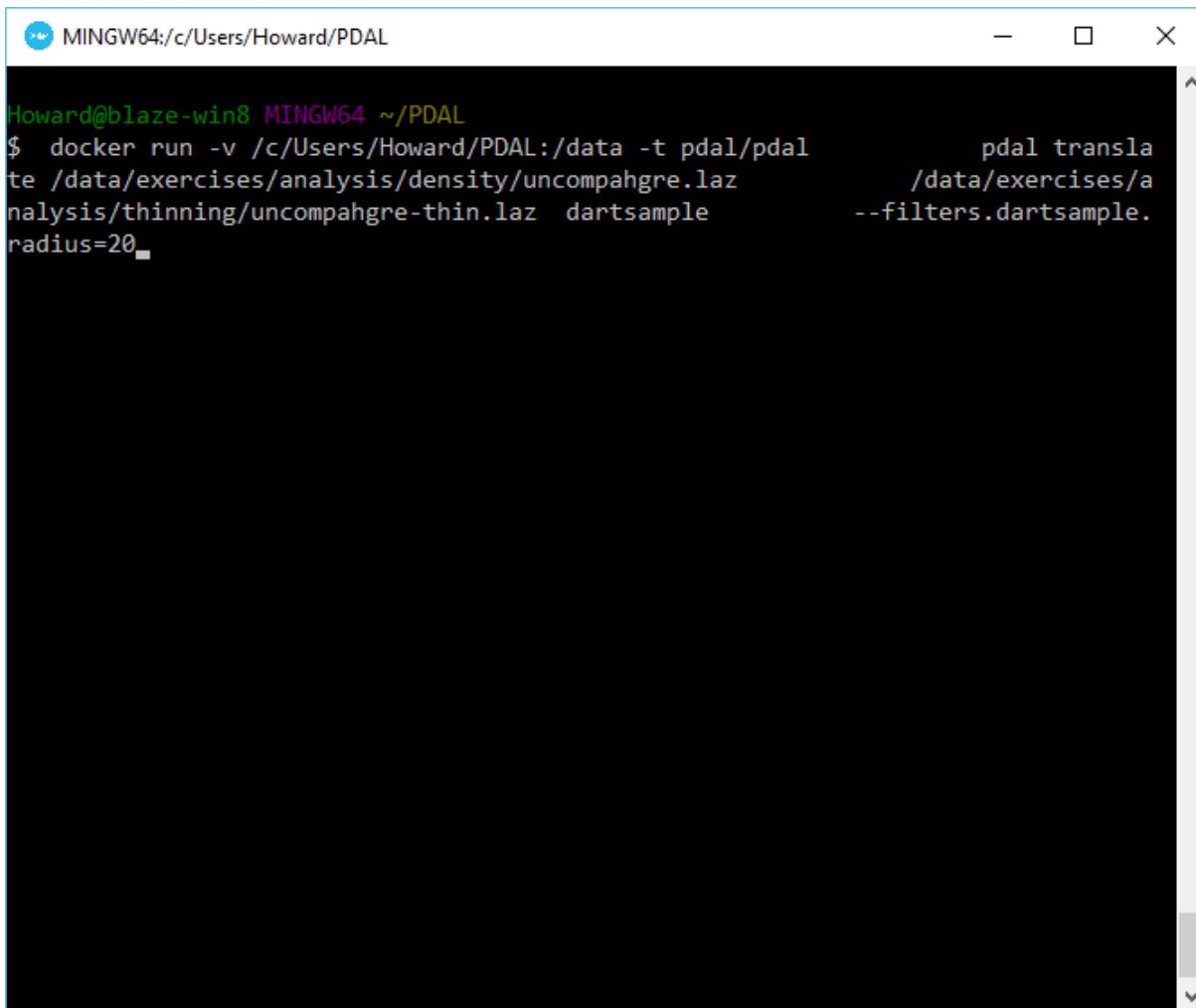


Fig. 8.4: Thinning strategies available in PDAL

A screenshot of a terminal window titled "MINGW64:/c/Users/Howard/PDAL". The window shows a command being run in a Docker container. The command is: \$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal pdal translate /data/exercises/analysis/density/uncompahgre.laz /data/exercises/analysis/thinning/uncompahgre-thin.laz --filters.dartsample radius=20. The terminal has a light blue background and white text.

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 154) will retain all attribute information.

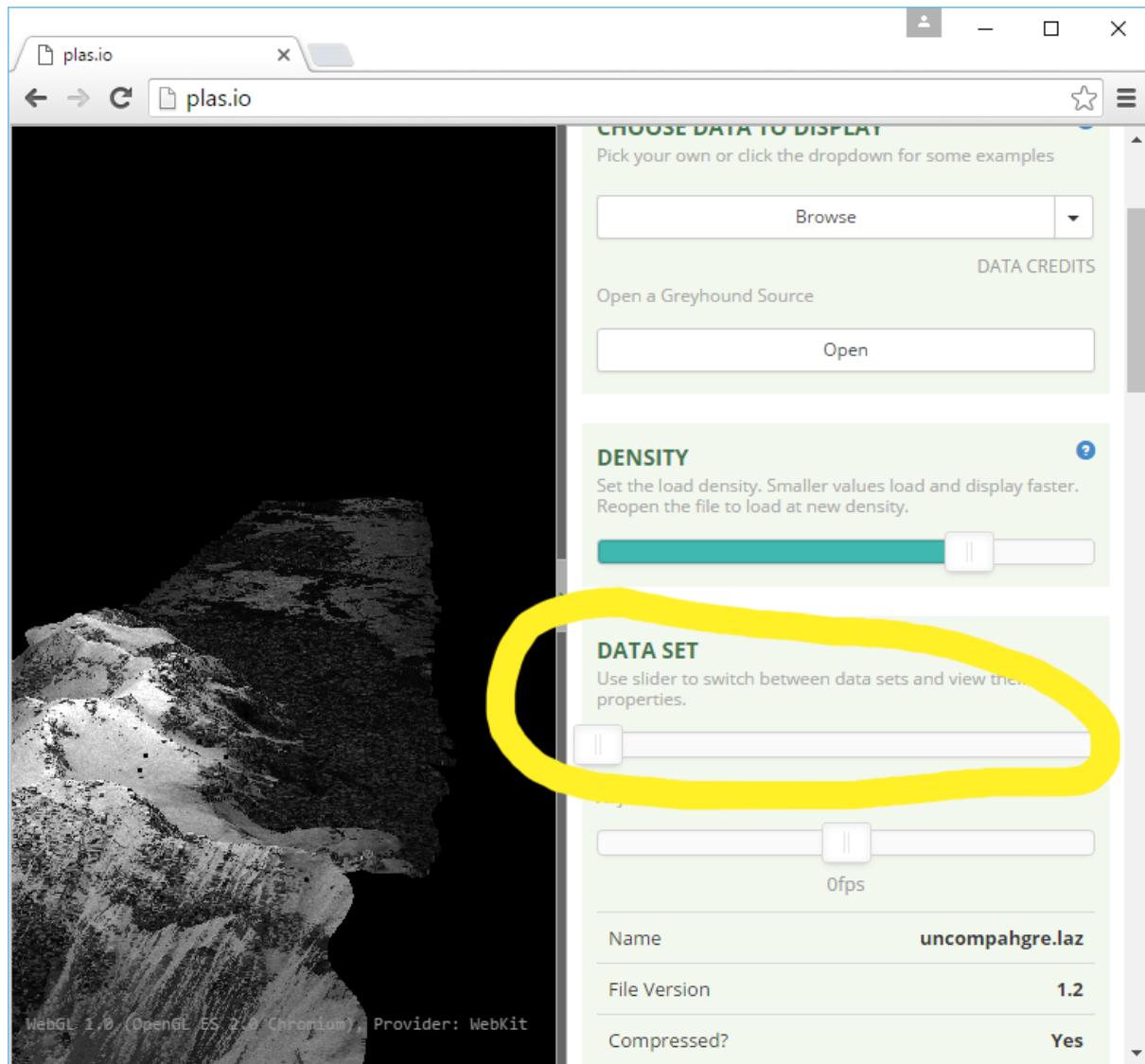


Fig. 8.5: Selecting multiple data sets in <http://plas.io>

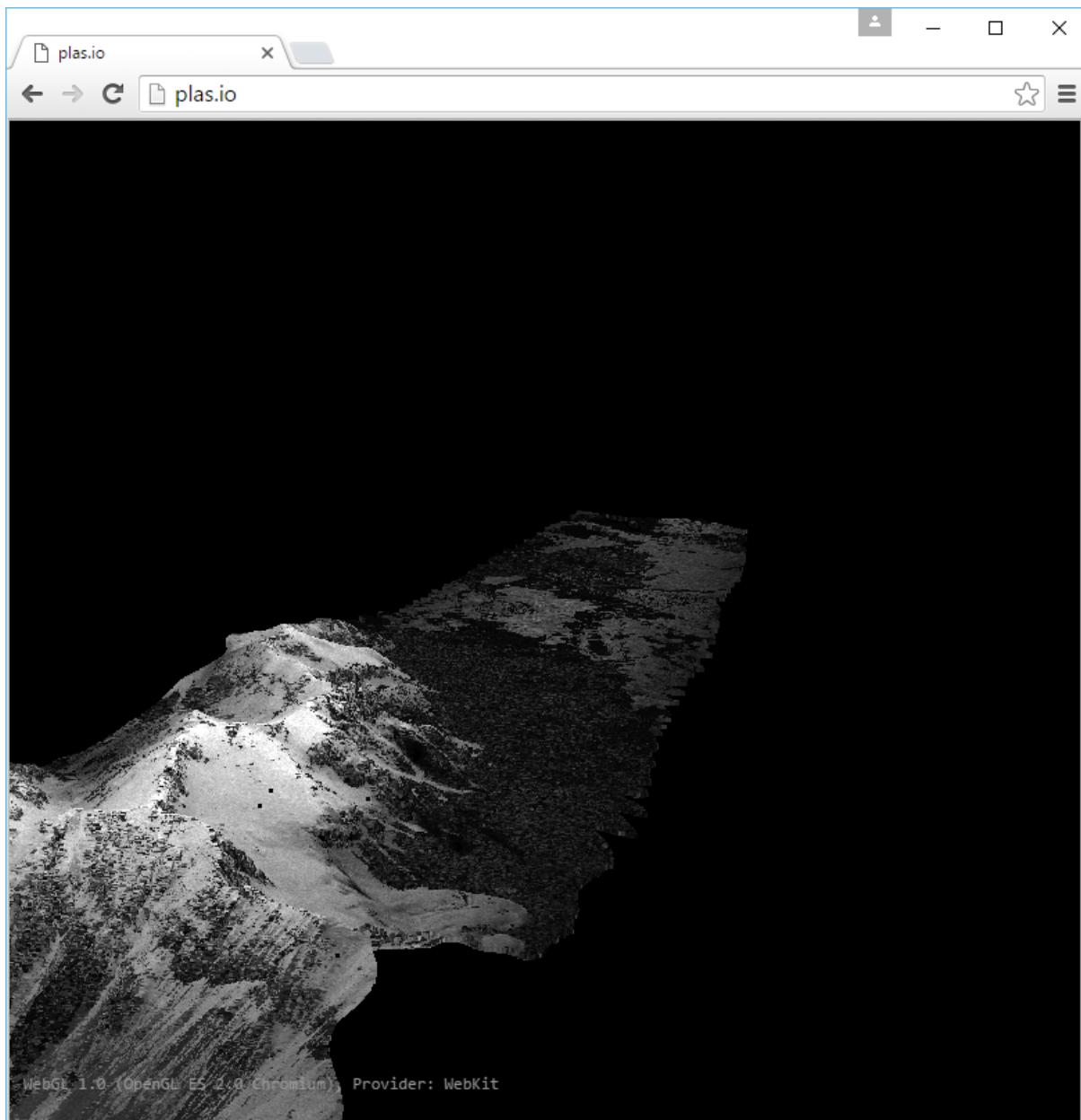


Fig. 8.6: Full resolution Uncompahgre data set

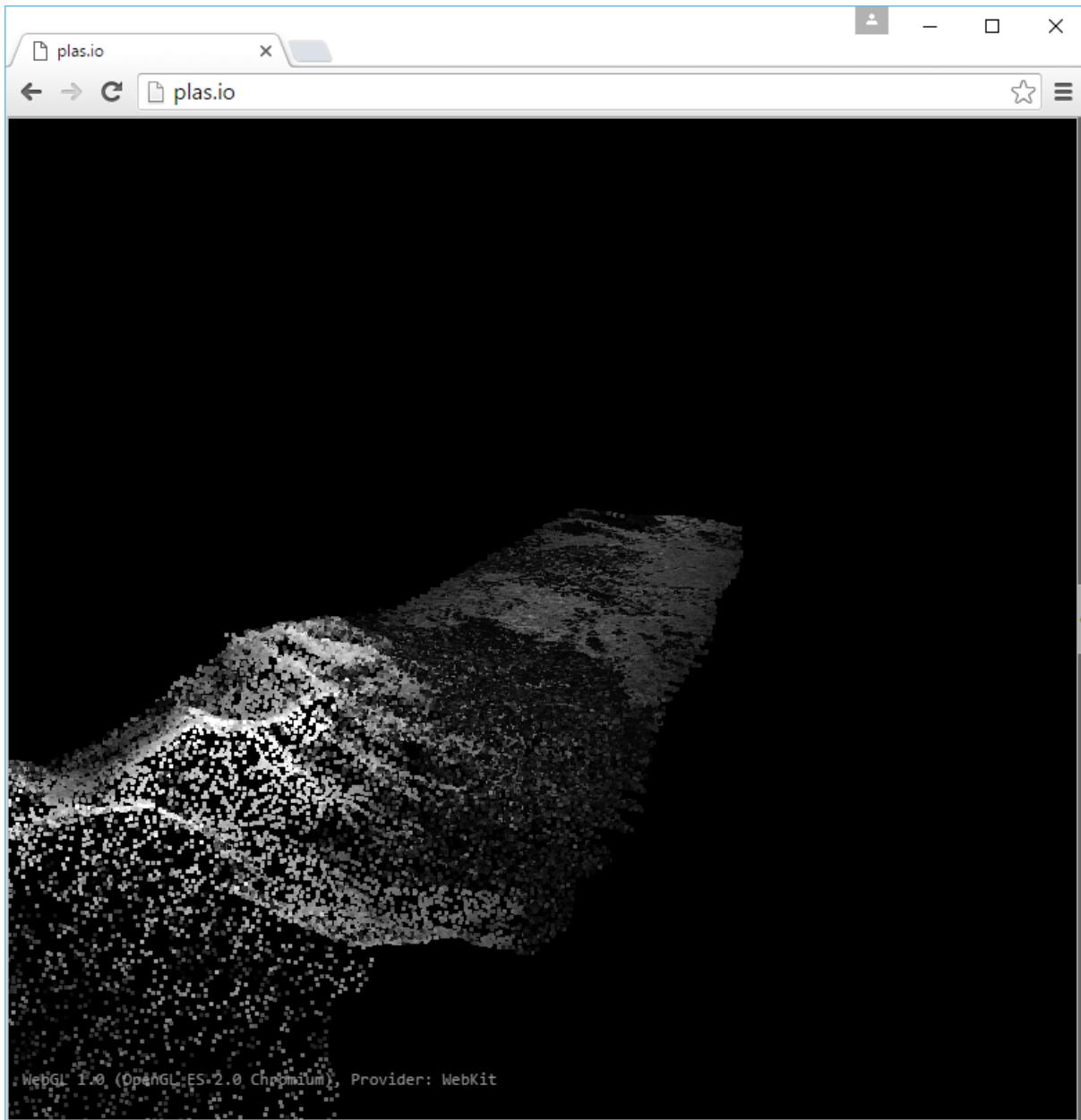


Fig. 8.7: Uncompahgre thinned at a radius of 20m

Identifying ground

This exercise uses PDAL to classify ground returns using the *Progressive Morphological Filter (PMF)* technique.

Note: This exercise is an adaptation of the *Identifying ground returns using ProgressiveMorphologicalFilter segmentation* (page 197) tutorial on the PDAL website by Brad Chambers. You can find more detail and example invocations there.

Exercise

The primary input for [Digital Terrain Model](#)

(https://en.wikipedia.org/wiki/Digital_elevation_model) generation is a point cloud with ground vs. not-ground classifications. In this example, we will use an algorithm provided by PDAL, the *Progressive Morphological Filter* technique to generate a ground surface.

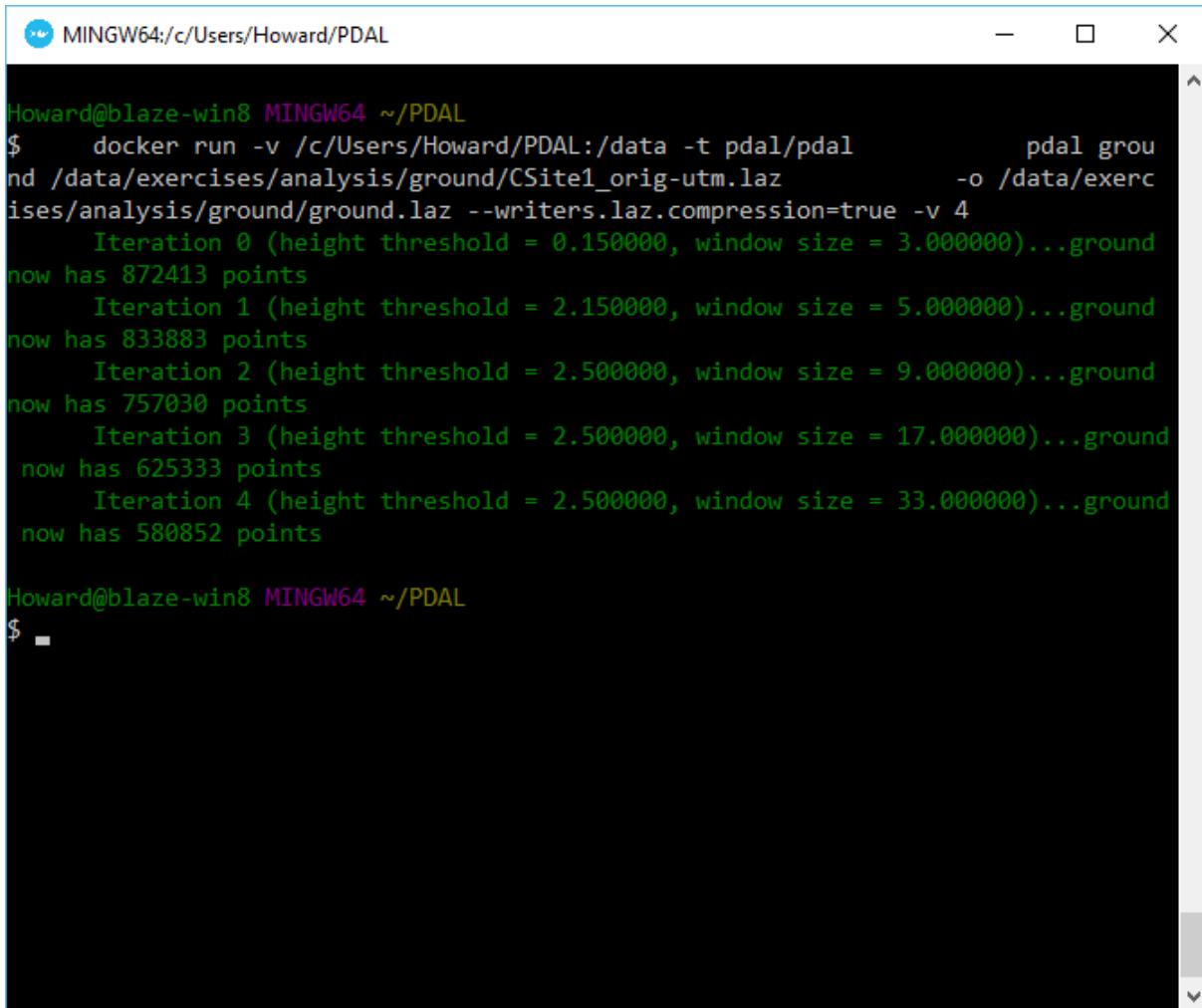
See also:

You can read more about the specifics of the PMF algorithm from the [paper](#) (<http://users.cis.fiu.edu/~chens/PDF/TGRS.pdf>), and you can read more about the PDAL implementation in the source code on [github](#) (<https://github.com/PDAL/PDAL/blob/master/filters/PMFFilter.cpp>).

Command

Invoke the following command, substituting accordingly, in your *Docker Quickstart Terminal*:

```
1 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
2     pdal ground \
3     /data/exercises/analysis/ground/CSitel_orig-utm.laz \
4     -o /data/exercises/analysis/ground/ground.laz \
5     --classify=true \
6     --writers.las.compression=true -v 4
```



The screenshot shows a terminal window titled "MINGW64:/c/Users/Howard/PDAL". The command run is:

```
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal pdal group /data/exercises/analysis/ground/CSite1_orig-utm.laz -o /data/exercises/analysis/ground.laz --writers.laz.compression=true -v 4
```

The output shows the iterative process of ground extraction:

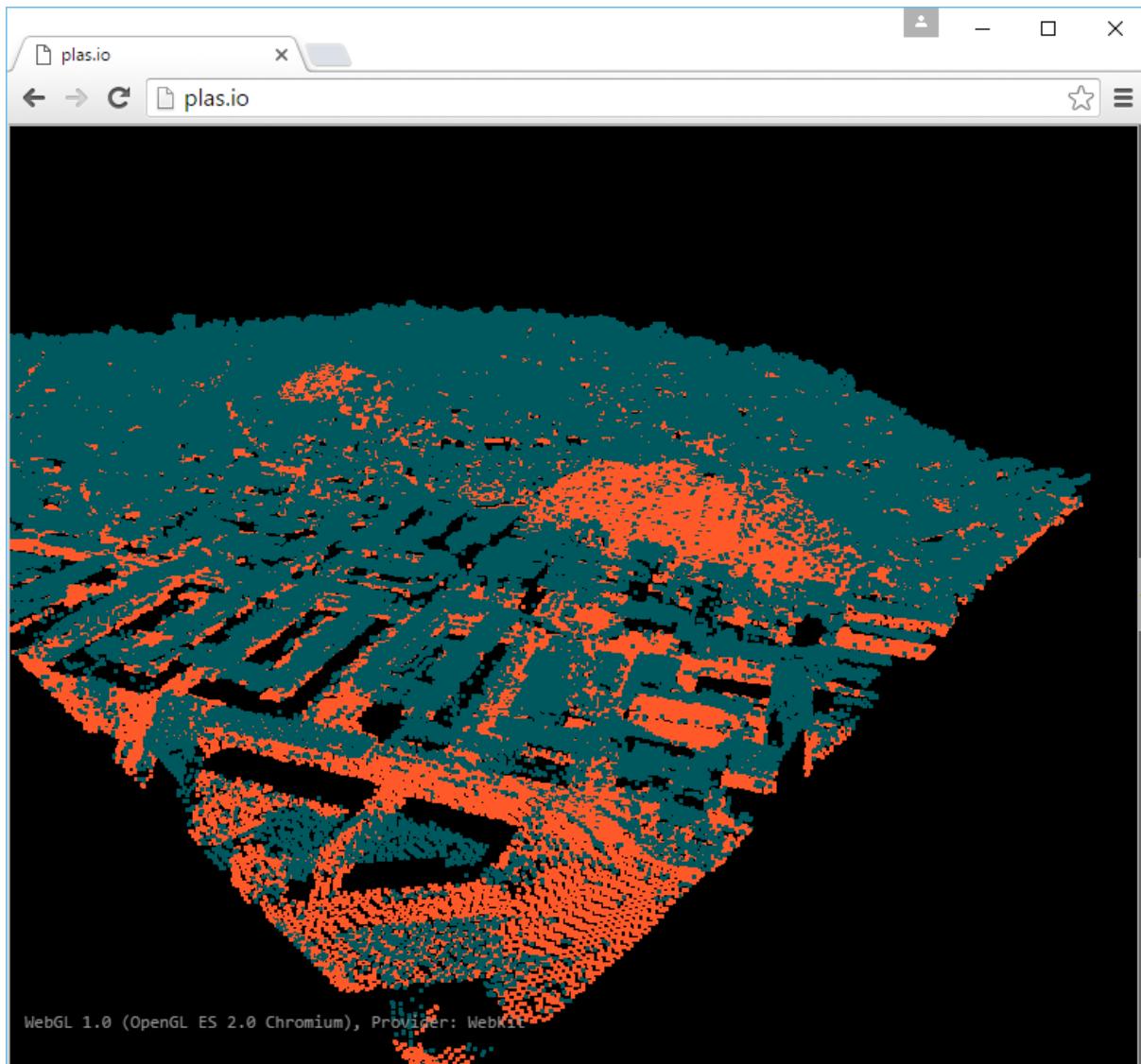
- Iteration 0 (height threshold = 0.150000, window size = 3.000000)...ground now has 872413 points
- Iteration 1 (height threshold = 2.150000, window size = 5.000000)...ground now has 833883 points
- Iteration 2 (height threshold = 2.500000, window size = 9.000000)...ground now has 757030 points
- Iteration 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

Howard@blaze-win8 MINGW64 ~/PDAL

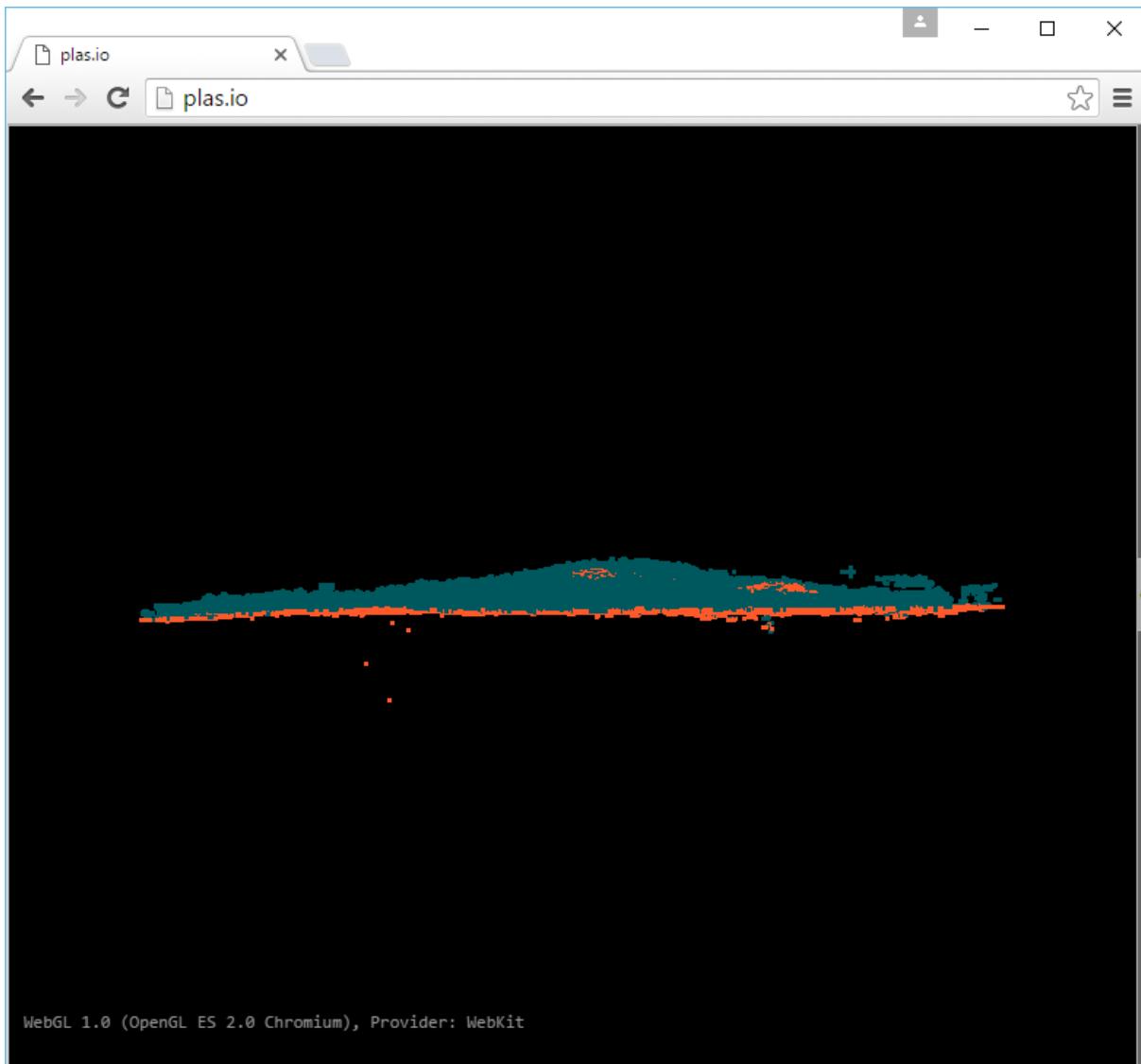
\$

As we can see, the algorithm does a great job of discriminating the points, but there's a few issues.

PDAL: Point cloud Data Abstraction Library, 1.4.0



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 [Removing noise](#) (page 323).

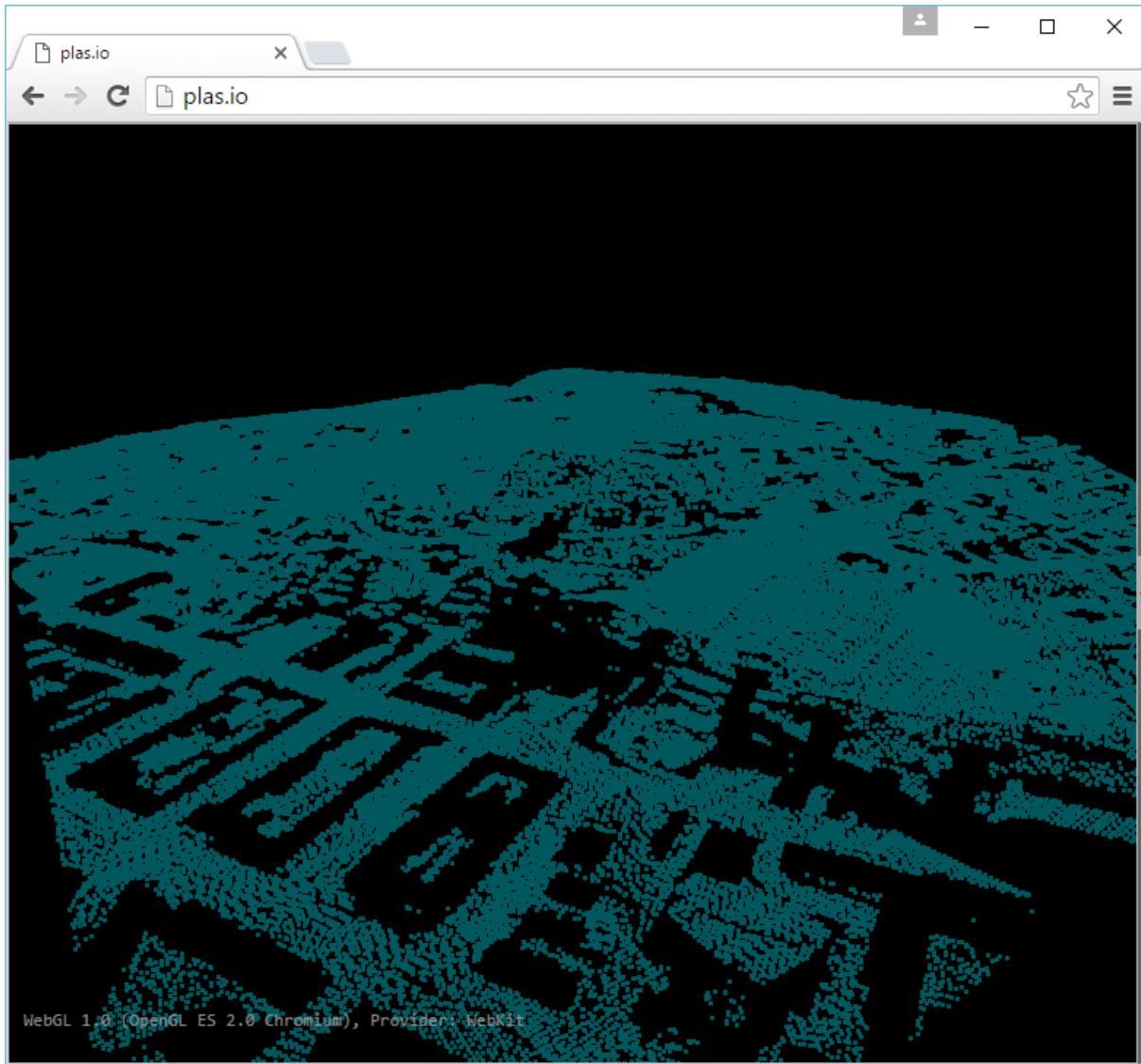
1. Let us start by removing the non-ground data:

PDAL: Point cloud Data Abstraction Library, 1.4.0

```
1 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
2     pdal ground \
3     /data/exercises/analysis/ground/CSite1_orig-utm.laz \
4     -o /data/exercises/analysis/ground/ground-only.laz \
5     --classify=true --extract=true \
6     --writers.las.compression=true --verbose 4
```

Note: The `filters.pmf.extract=true` item causes all data except ground-classified points to be removed from the set.

Buildings and other non-ground points are removed with the `extract` option of *filters.pmf* (page 144)



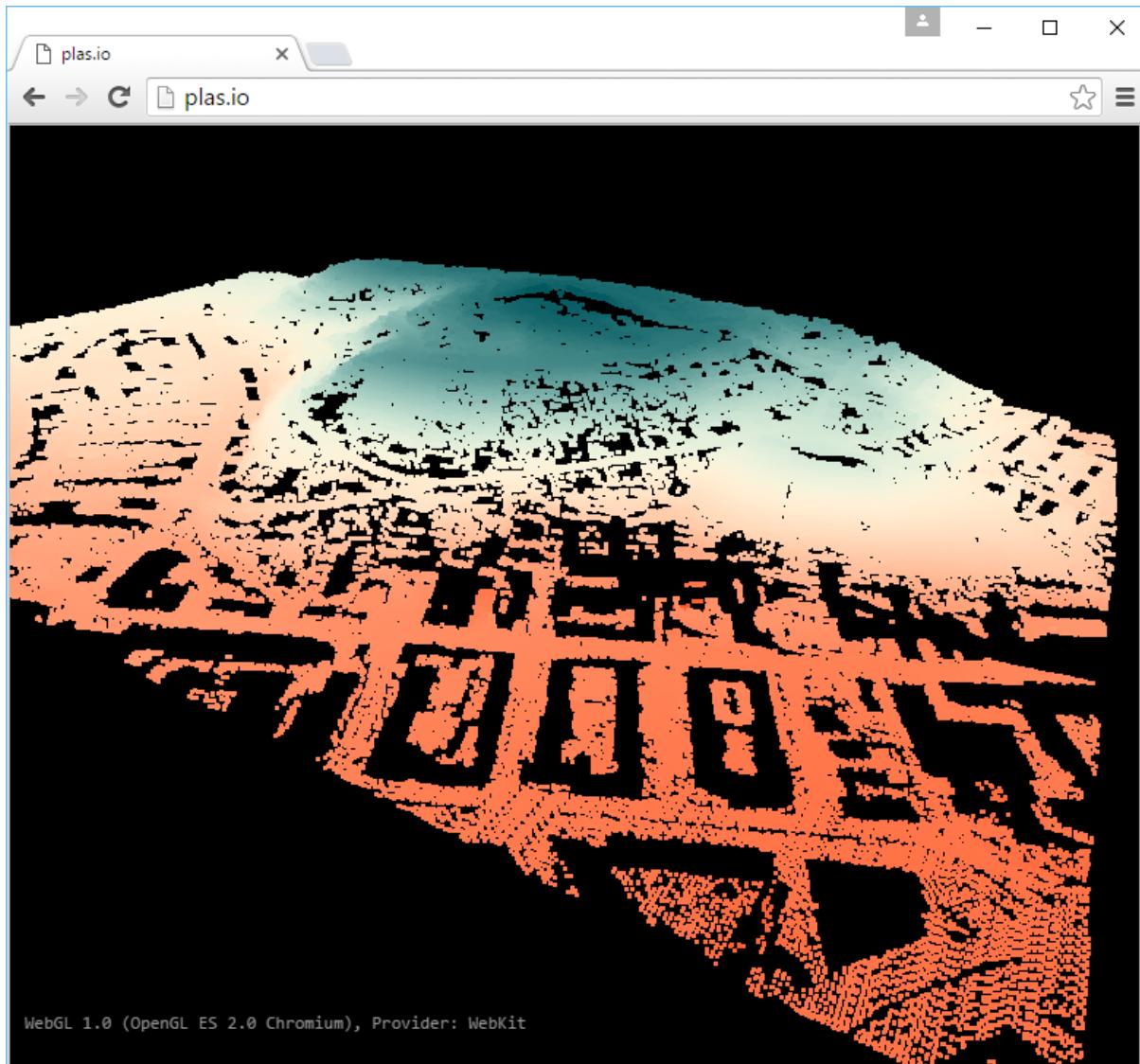
2. Now we will remove the noise, using the *translate* (page 32) to stack the *filters.outlier* (page 137) and *filters.pmf* (page 144) stages:

```
1 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
2     pdal translate \
3     /data/exercises/analysis/ground/CSitel_orig-utm.laz \
4     -o /data/exercises/analysis/ground/denoised-ground-only.laz \
5     outlier pmf \
6     --filters.outlier.method="statistical" \
```

PDAL: Point cloud Data Abstraction Library, 1.4.0

```
7      --filters.outlier.mean_k=8 \
8      --filters.outlier.multiplier=3.0 \
9      --filters.pmf.cell_size=1.5 \
10     --filters.pmf.extract=true \
11     --writers.las.compression=true --verbose 4
```

The result is a more accurate representation of the ground returns.



Generating a DTM

This exercise uses PDAL to generate an elevation model surface using the output from the [Identifying ground](#) (page 342) exercise, PDAL's [writers.p2g](#) (page 92) 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 `ground-filtered.laz`, in the [Identifying ground](#) (page 342) exercise. Please produce that file by following that exercise before starting this one.

Command

Invoke the following command, substituting accordingly, in your *Docker Quickstart Terminal*:
PDAL capability to generate rasterized output is provided by the [writers.p2g](#) (page 92) stage.
There is no [application](#) (page 17) to drive this stage, and we must use a pipeline.

Pipeline breakdown

```
{  
    "pipeline": [  
        "/data/exercises/analysis/ground/ground-filtered.laz",  
        {  
            "filename": "/data/exercises/analysis/dtm/dtm",  
            "output_format": "tif",  
            "output_type": "all",  
            "grid_dist_x": "2.0",  
            "grid_dist_y": "2.0",  
            "type": "writers.p2g"  
        }  
    ]  
}
```

```
    ]  
}
```

Note: this pipeline is available in your workshop materials in the `./exercises/analysis/dtm/dtm.json` file.

1. Reader

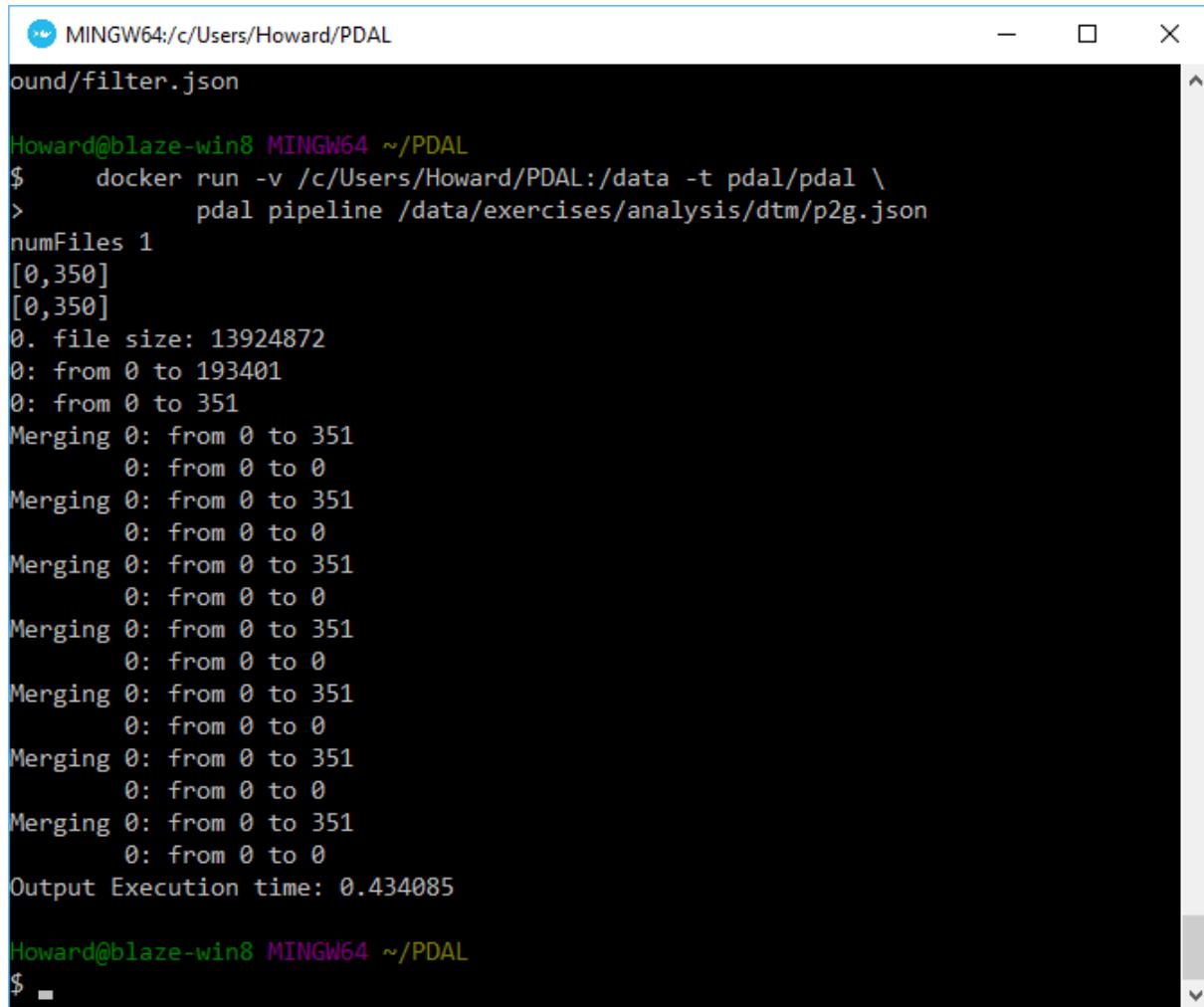
`ground-filtered.laz` is the [LASzip](http://laszip.org) (<http://laszip.org>) file we will clip. You should have created this output as part of the *Identifying ground* (page 342) exercise.

2. writers.p2g

The [Points2grid](https://github.com/CRREL/points2grid) (<https://github.com/CRREL/points2grid>) writer that bins the point cloud data into an elevation surface.

Execution

```
1 docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \  
2     pdal pipeline \  
3     /data/exercises/analysis/dtm/p2g.json
```



The screenshot shows a terminal window titled "MINGW64:/c/Users/Howard/PDAL". The command run is:

```
Howard@blaze-win8 MINGW64 ~/PDAL
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
>     pdal pipeline /data/exercises/analysis/dtm/p2g.json
```

The output of the command is:

```
numFiles 1
[0,350]
[0,350]
0. file size: 13924872
0: from 0 to 193401
0: from 0 to 351
Merging 0: from 0 to 351
    0: from 0 to 0
Merging 0: from 0 to 351
    0: from 0 to 0
Merging 0: from 0 to 351
    0: from 0 to 0
Merging 0: from 0 to 351
    0: from 0 to 0
Merging 0: from 0 to 351
    0: from 0 to 0
Merging 0: from 0 to 351
    0: from 0 to 0
Merging 0: from 0 to 351
    0: from 0 to 0
Merging 0: from 0 to 351
    0: from 0 to 0
Output Execution time: 0.434085
```

At the bottom of the terminal window, there is a prompt:

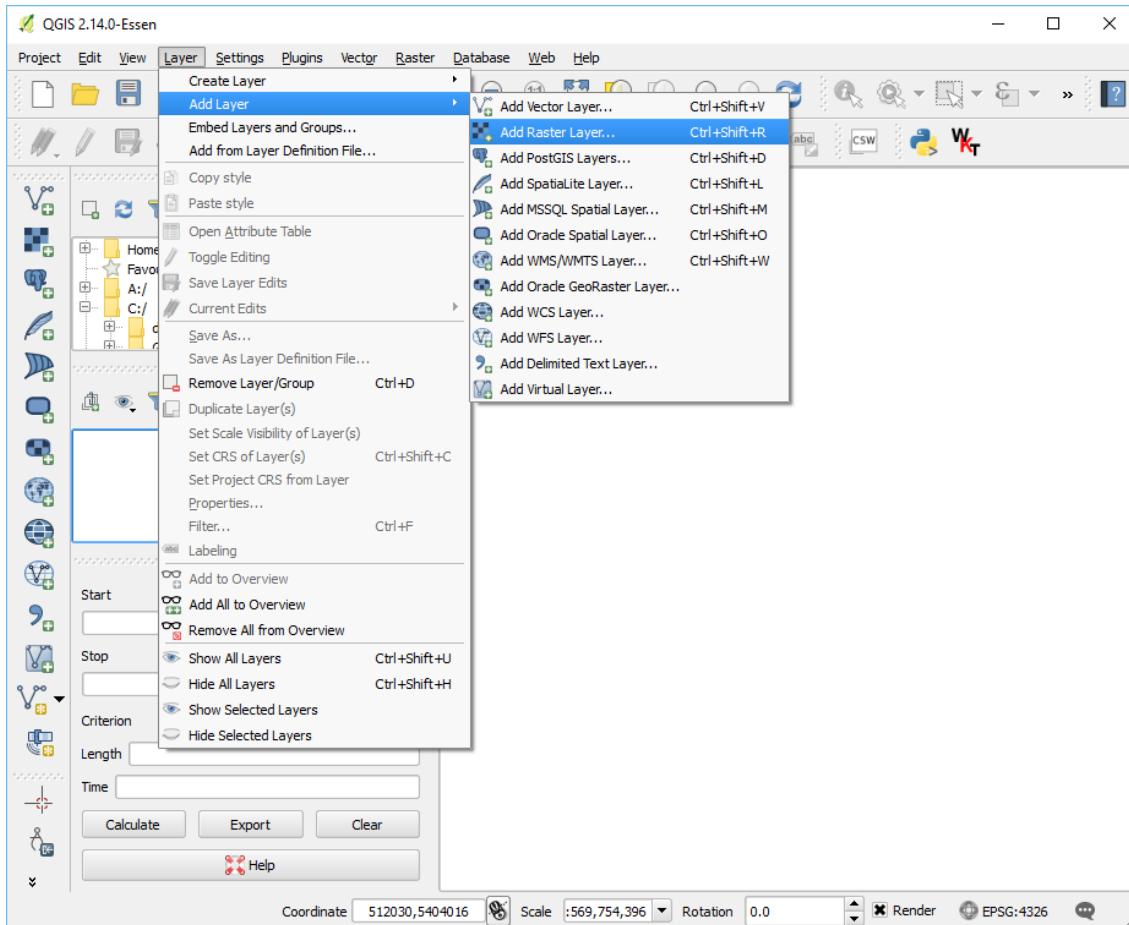
```
Howard@blaze-win8 MINGW64 ~/PDAL
$
```

Visualization

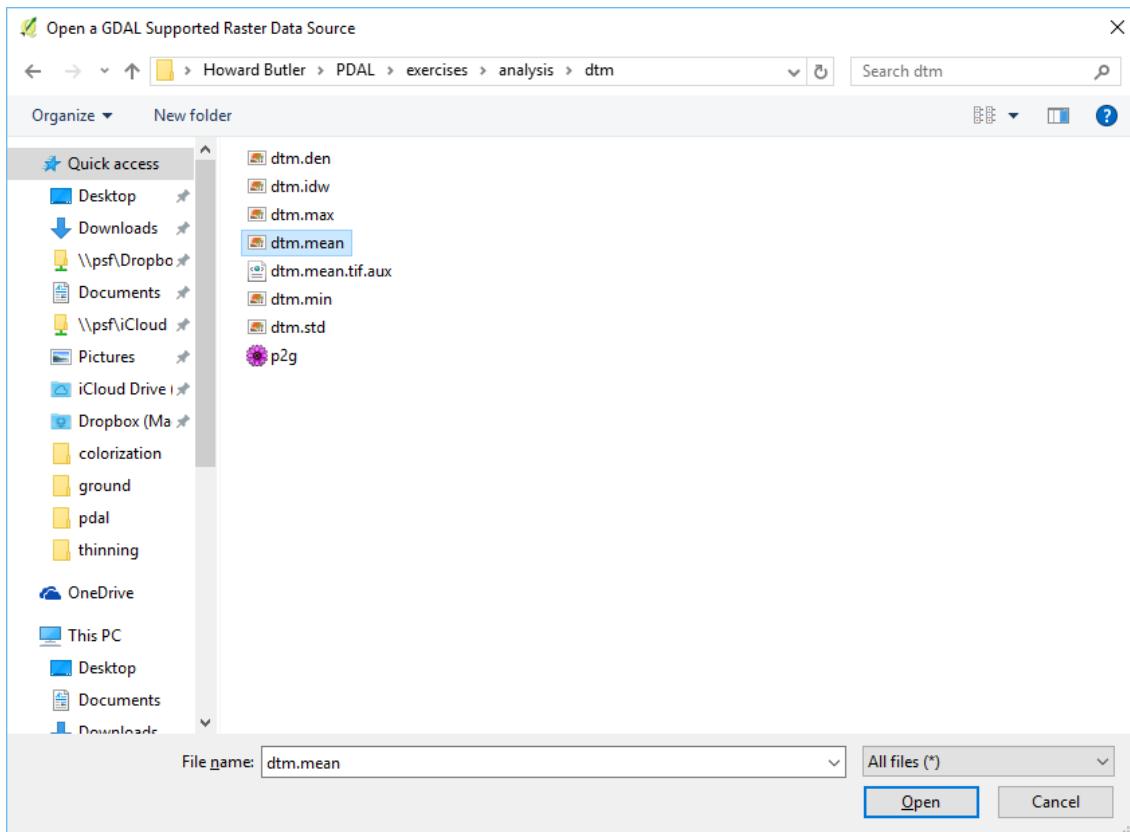
Something happened, and some files were written, but we cannot really see what was produced. Let us use [QGIS](#) (page 286) to visualize the output.

1. Open [QGIS](#) (page 286) and *Add Raster Layer*:

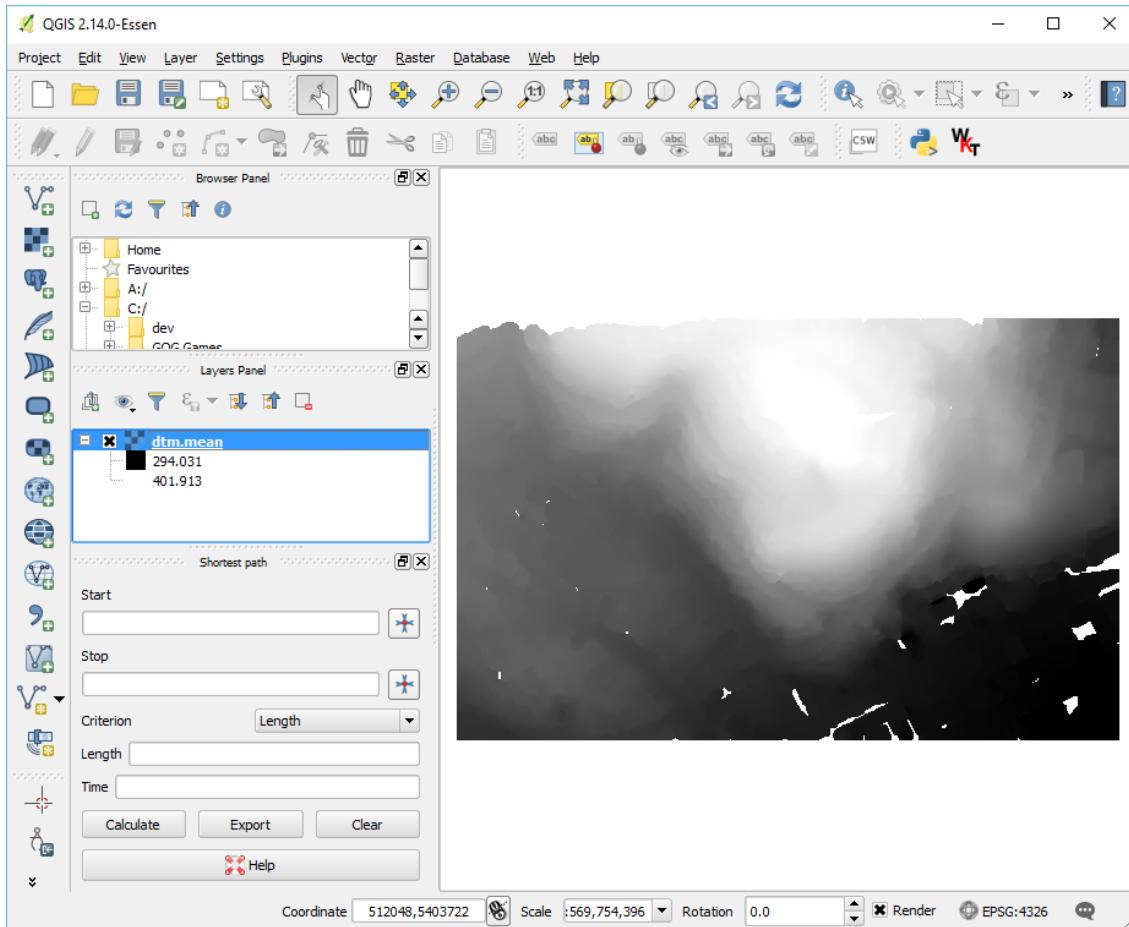
PDAL: Point cloud Data Abstraction Library, 1.4.0



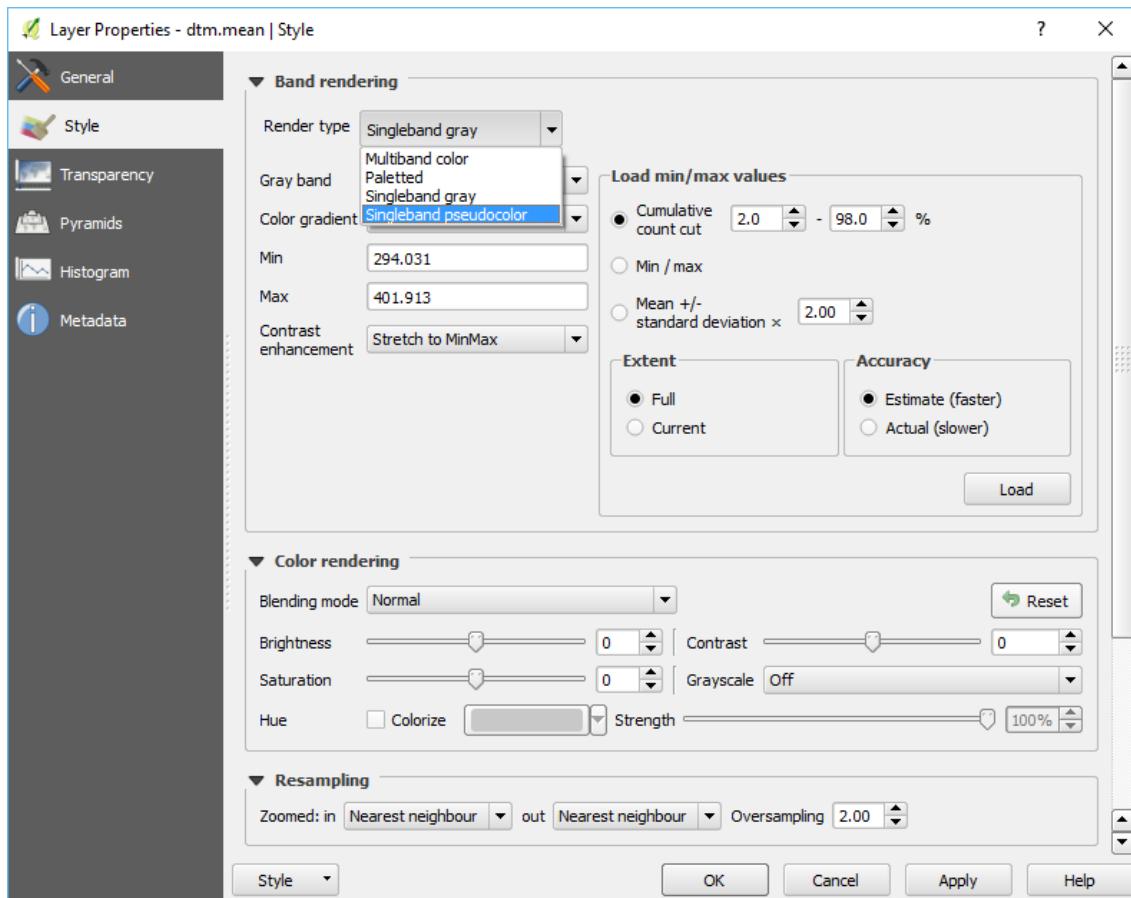
2. Add the *dtm.idw.tif* file from your `./PDAL/exercises/analysis/dtm` directory.



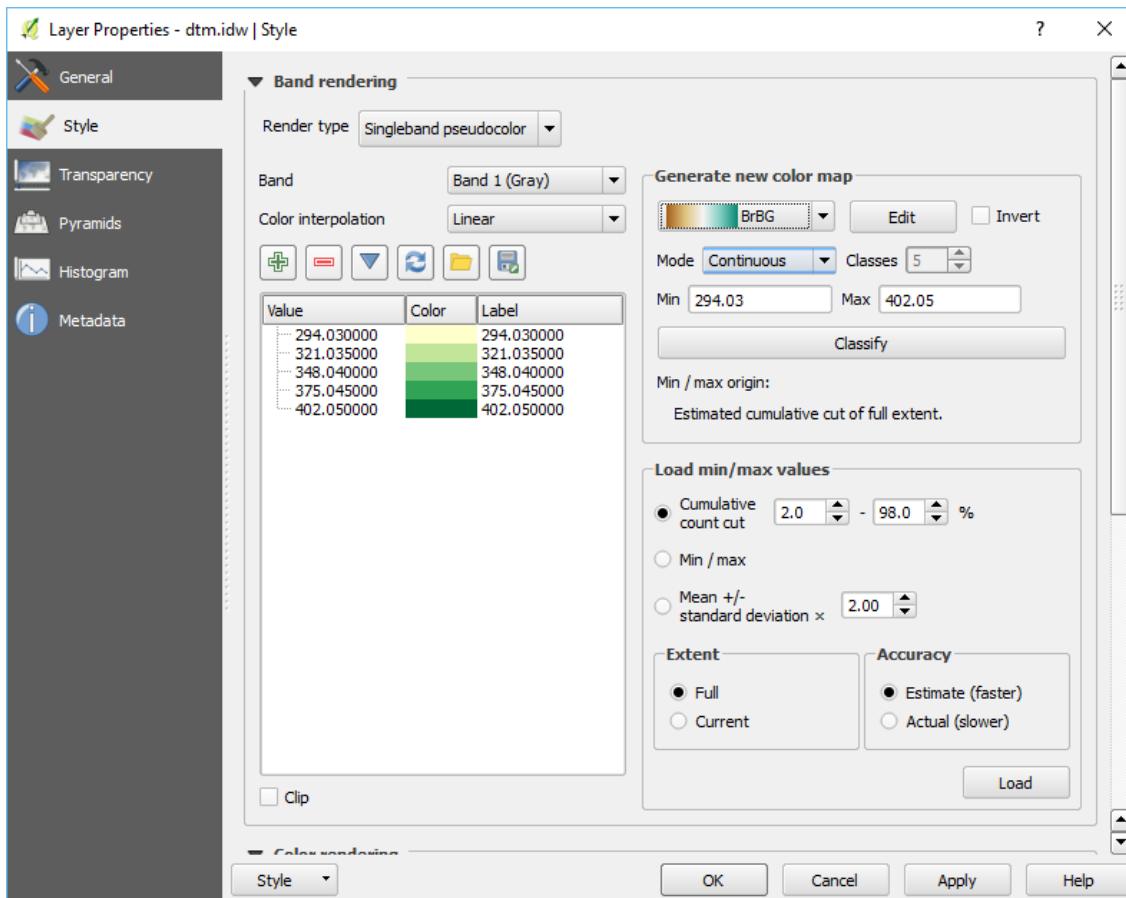
PDAL: Point cloud Data Abstraction Library, 1.4.0



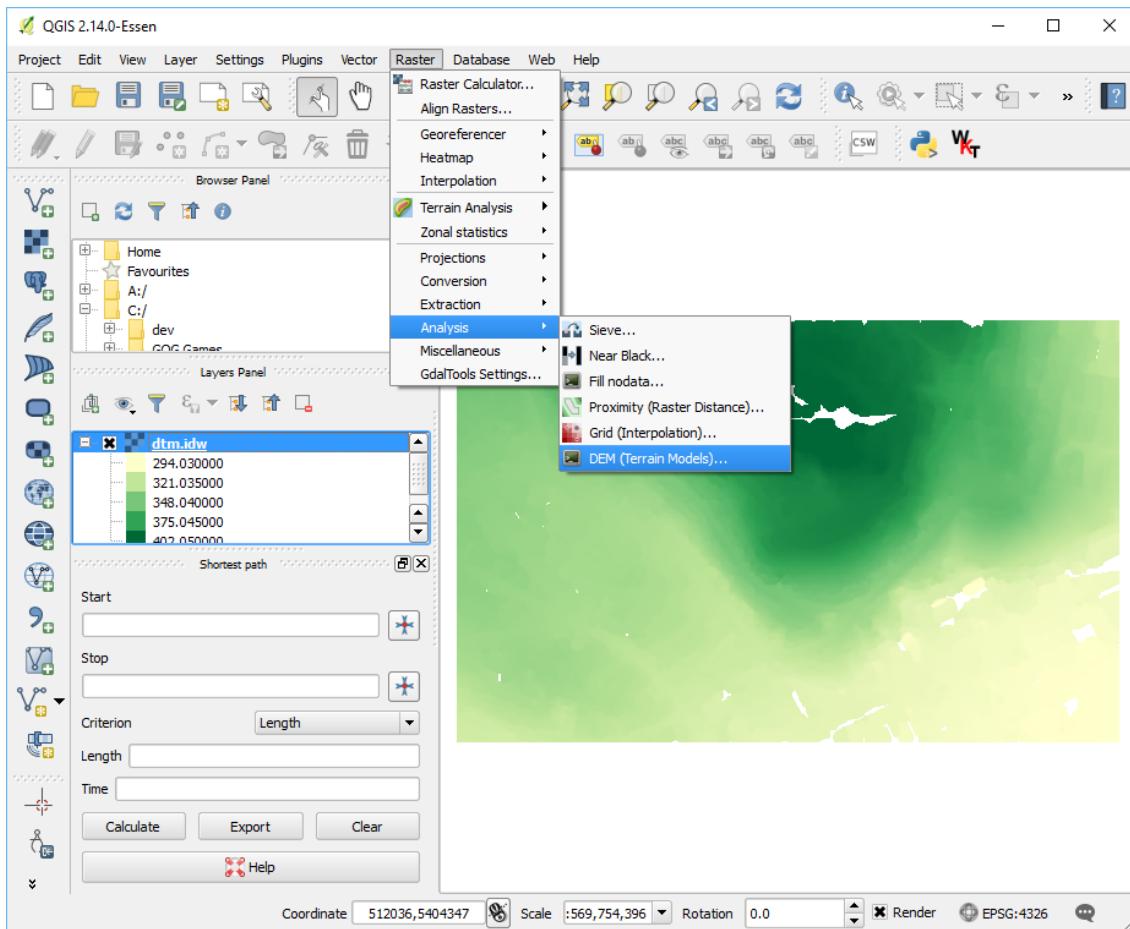
3. Classify the DTM by right-clicking on the *dtm.idw.tif* and choosing *Properties*. Pick the pseudocolor rendering type, and then choose a color ramp and click *Classify*.



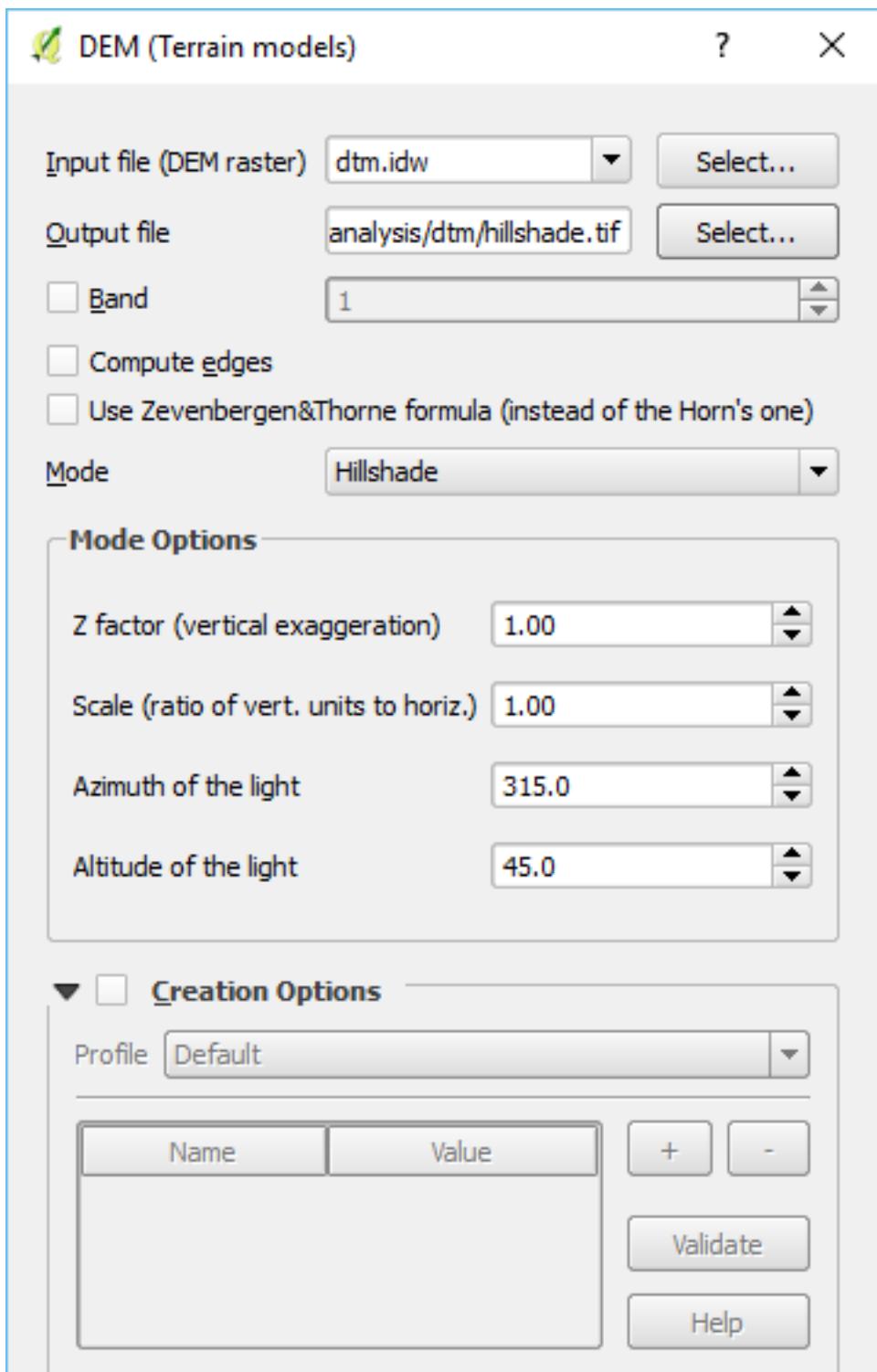
PDAL: Point cloud Data Abstraction Library, 1.4.0



4. *QGIS* (page 286) provides access to *GDAL* (<http://gdal.org/>) processing tools, and we are going to use that to create a hillshade of our surface. Choose *Raster->Analysis->Dem*:



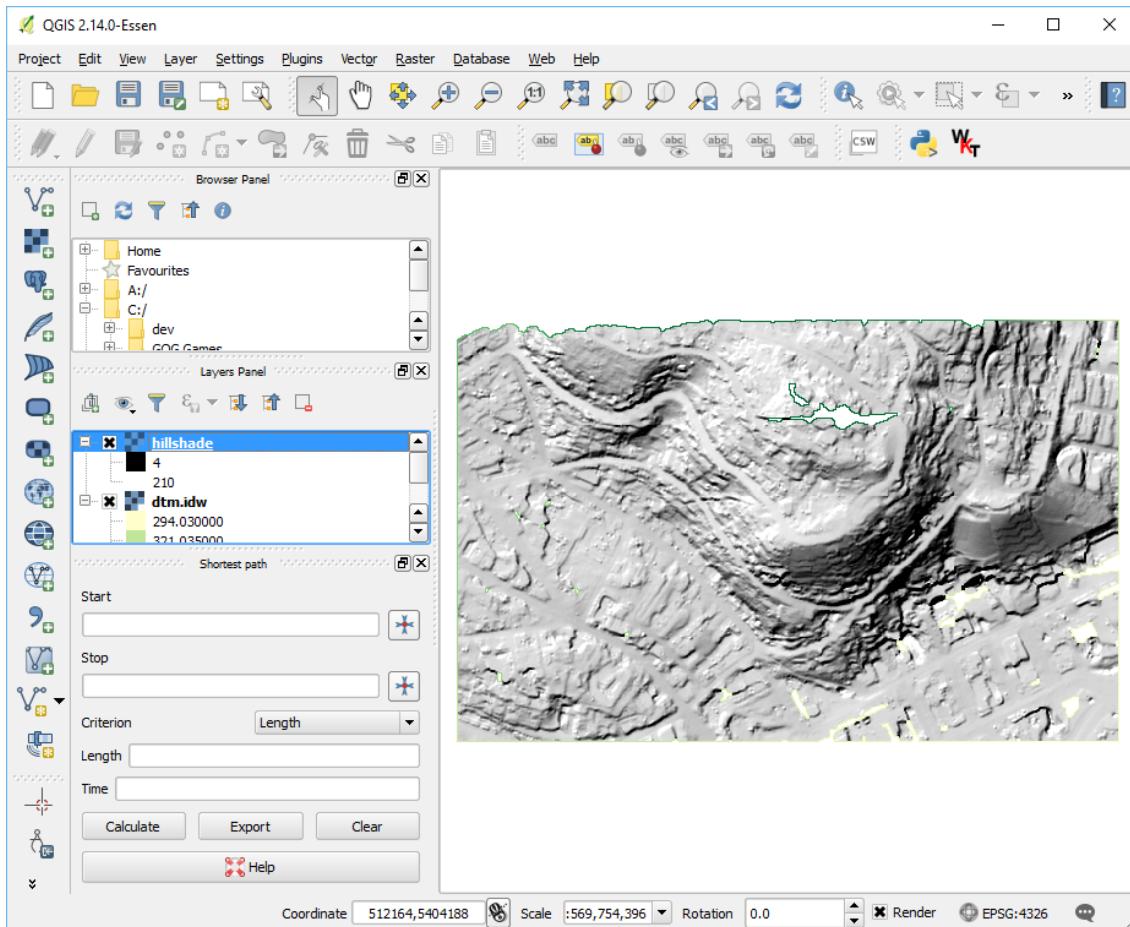
5. Click the window for the *Output file* and select a location to save the `hillshade.tif` file.



```
gdaldem hillshade C:  
\\Users\\Howard\\PDAL\\exercises\\analysis\\dtm\\dtm.idw.tif  
C:\\Users\\Howard\\PDAL\\exercises\\analysis\\dtm\\hillshade.tif -  
z 1.0 -s 1.0 -az 315.0 -alt 45.0 -of GTiff
```



- Click *OK* and the hillshade of your DTM is now available



Notes

- `gdaldem` (<http://www.gdal.org/gdaldem.html>), which powers the *QGIS* (page 286) DEM tools, is a very powerful command line utility you can use for processing data.
- `Points2grid` (<https://github.com/CRREL/points2grid>) can be used for large data, but it does not interpolate typical `TIN` (https://en.wikipedia.org/wiki/Triangulated_irregular_network) surface model before interpolating.

Georeferencing

Georeferencing

As discussed *in the introduction* (page 269), 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 151) PDAL filter to drop all points with very high intensity values.

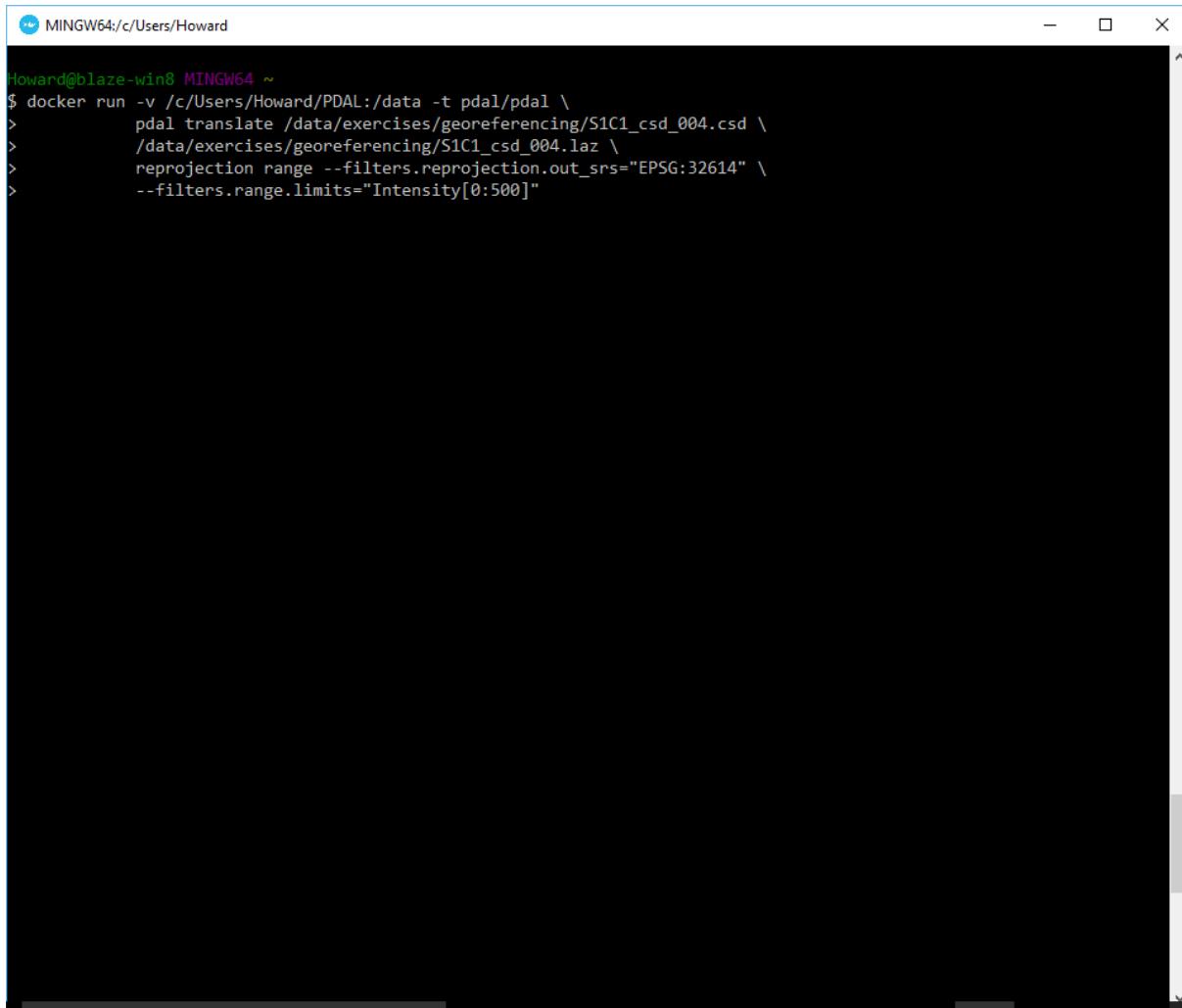
Note: These data were provided by Dr. Craig Glennie and were collected by [NCALM](#) (<http://ncalm.cive.uh.edu/>), the National Center for Airborne Laser Mapping. The collect area is southwest of Austin, TX.

Command

Invoke the following command, substituting accordingly, into your *Docker Quickstart Terminal*:

```
docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
    pdal translate \
    /data/exercises/georeferencing/S1C1_csd_004.csd \
    /data/exercises/georeferencing/S1C1_csd_004.laz \
    reprojection range \
    --filters.reprojection.out_srs="EPSG:32614" \
    --filters.range.limits="Intensity[0:500]"
```

PDAL: Point cloud Data Abstraction Library, 1.4.0

A screenshot of a terminal window titled "MINGW64:/c/Users/Howard". The window contains a command-line interface for the PDAL library. The command being run is:

```
Howard@blaze-win8 MINGW64 ~
$ docker run -v /c/Users/Howard/PDAL:/data -t pdal/pdal \
>     pdal translate /data/exercises/georeferencing/S1C1_csd_004.csd \
>     /data/exercises/georeferencing/S1C1_csd_004.laz \
>     reprojection range --filters.reprojection.out_srs="EPSG:32614" \
>     --filters.range.limits="Intensity[0:500]"
```

Visualization

View your georeferenced point cloud in <http://plas.io>.

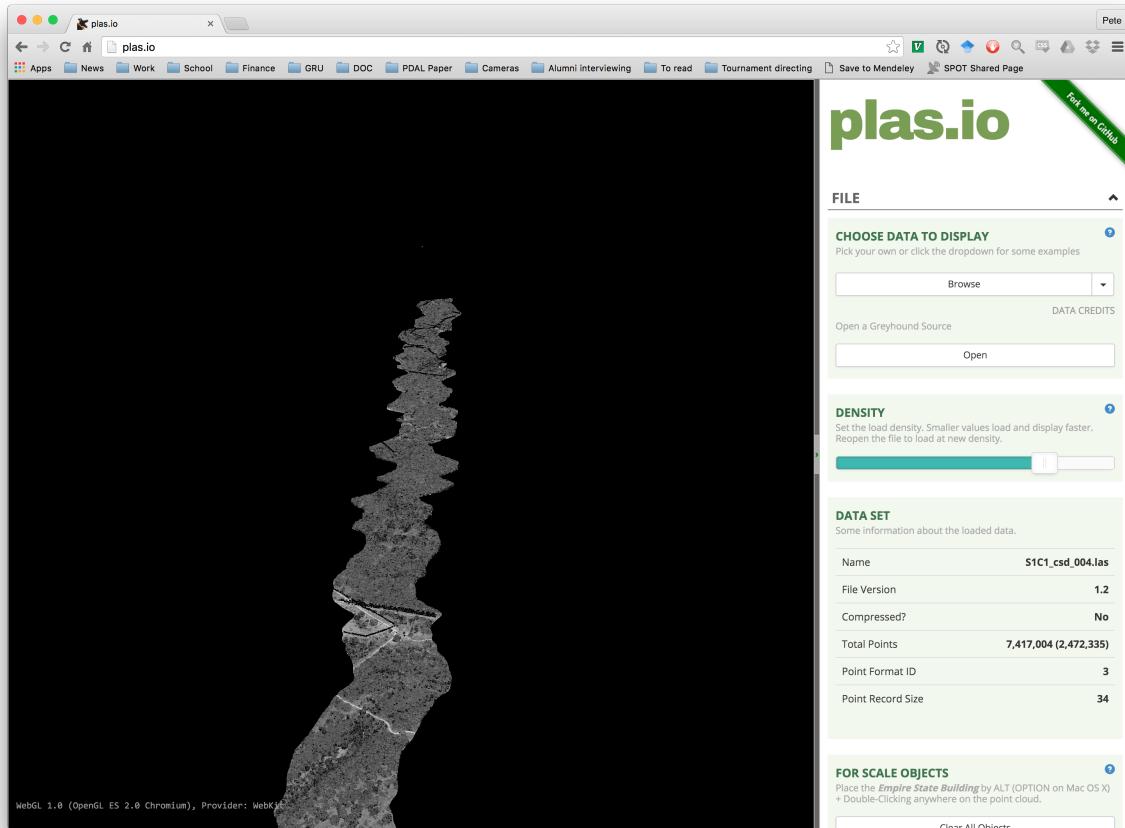


Fig. 8.8: Our airborne laser point cloud after georeferencing, reprojection, and intensity filtering.

**CHAPTER
NINE**

DEVELOPMENT

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.

Compilation

This section describes how to build and install PDAL under Windows, Linux, and Mac. PDAL's numerous *Dependencies* (page 390) can make it a challenge to build a fully-featured build.

See also:

Download (page 5) contains links to installable binaries for Windows, OSX, and RHEL Linux systems.

See also:

Install Docker (page 9) 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 17).

Contents:

Unix Compilation

Author Howard Butler

Contact howard@hobu.co

Date 10/27/2015

[CMake](http://www.cmake.org/) (<http://www.cmake.org/>) 2.8.11+ is the prescribed tool for building from source, with [CMake](http://www.cmake.org/) (<http://www.cmake.org/>) 3.0+ being desired. [CMake](http://www.cmake.org/) (<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/) (<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 9) contains an automated way to build PDAL and all of its dependencies.

Note: [Dependencies](#) (page 390) 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 6) for how to obtain the latest development version or visit [Download](#) (page 5) 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 Testing command

`make install` will install the *utilities* (page 17) 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/index.html)	ORA-CLE_INCLUDE_DIR	/home/oracle/sdk/include
	ORA-CLE_NNZ_LIBRARY	/home/oracle/libnnz10.so
	ORA-CLE_OCCI_LIBRARY	/home/oracle/libocci1.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 35) if you run into any issues.

While [CMake](#) (<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:

- 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 17) programs

```
make install
```

Using “XCode” on OS X

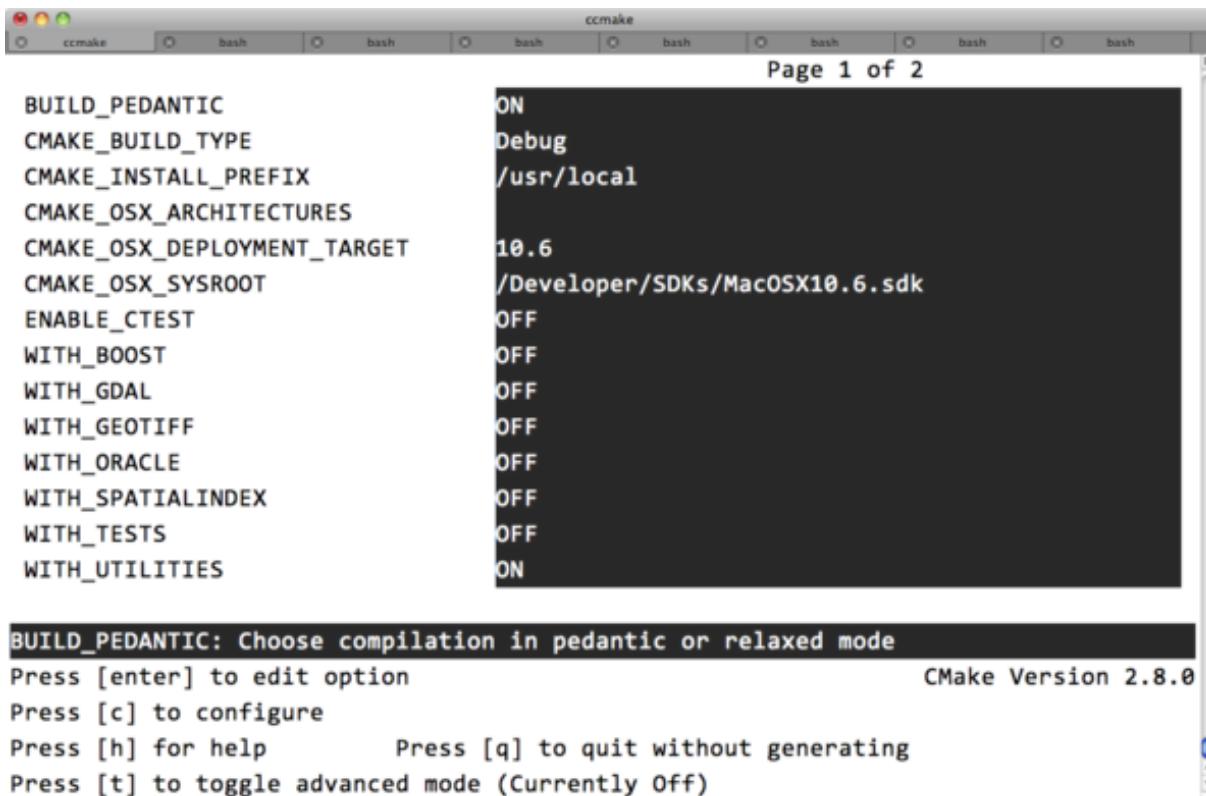


Fig. 9.1: Running the [Curses](http://en.wikipedia.org/wiki/Curses_%28programming_library%29) (http://en.wikipedia.org/wiki/Curses_%28programming_library%29) [CMake](http://www.cmake.org/) (<http://www.cmake.org/>) interface. This interface is available to all unix-like operating systems.

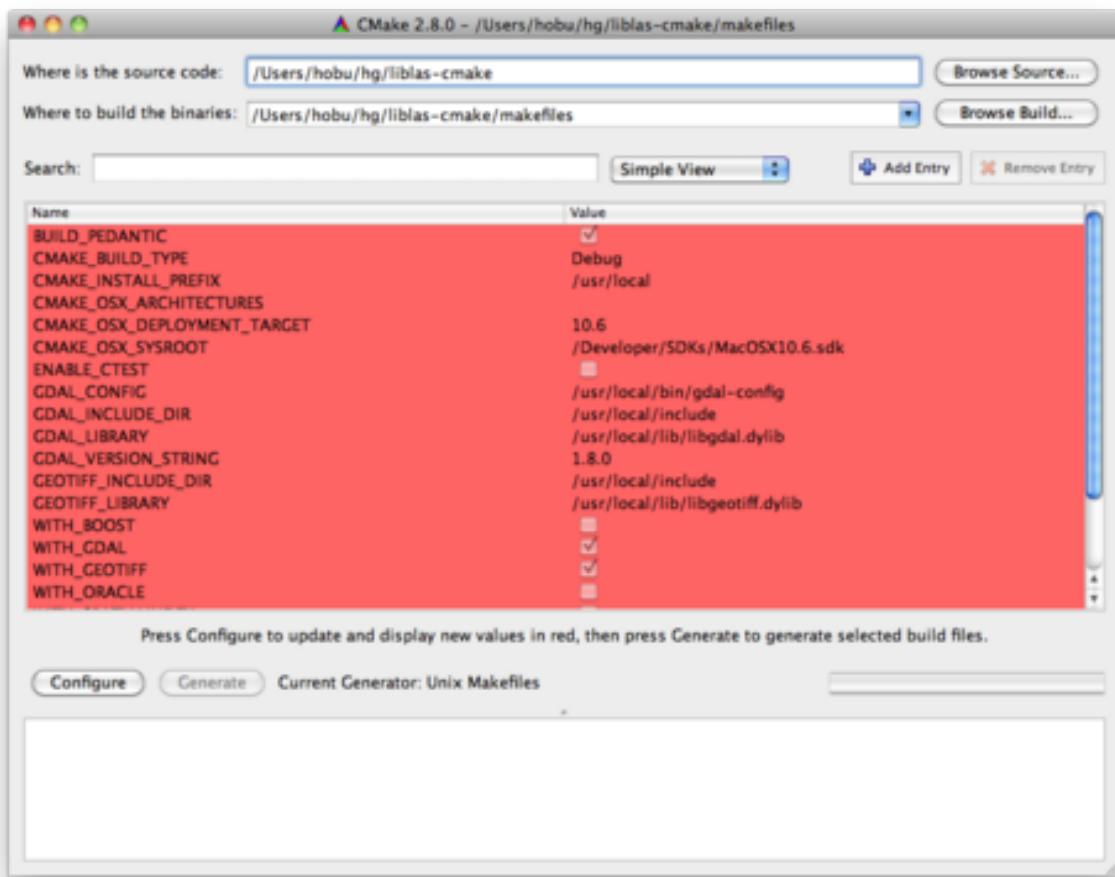


Fig. 9.2: Running the cmake-gui CMake (<http://www.cmake.org/>) interface. This interface is available on Linux, Windows, and Mac OS X.

Get the source code

See [Development Source](#) (page 6) for how to obtain the latest development version or visit [Download](#) (page 5) 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.

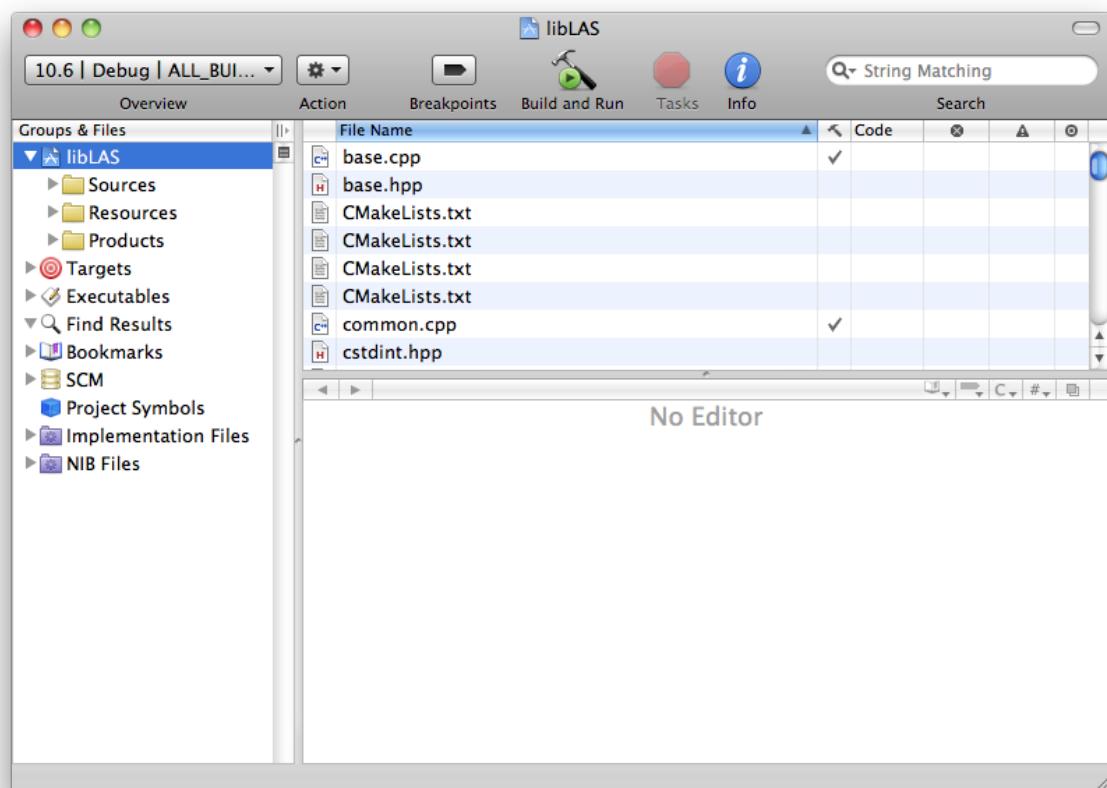
```
$ open PDAL.xcodeproj/
```

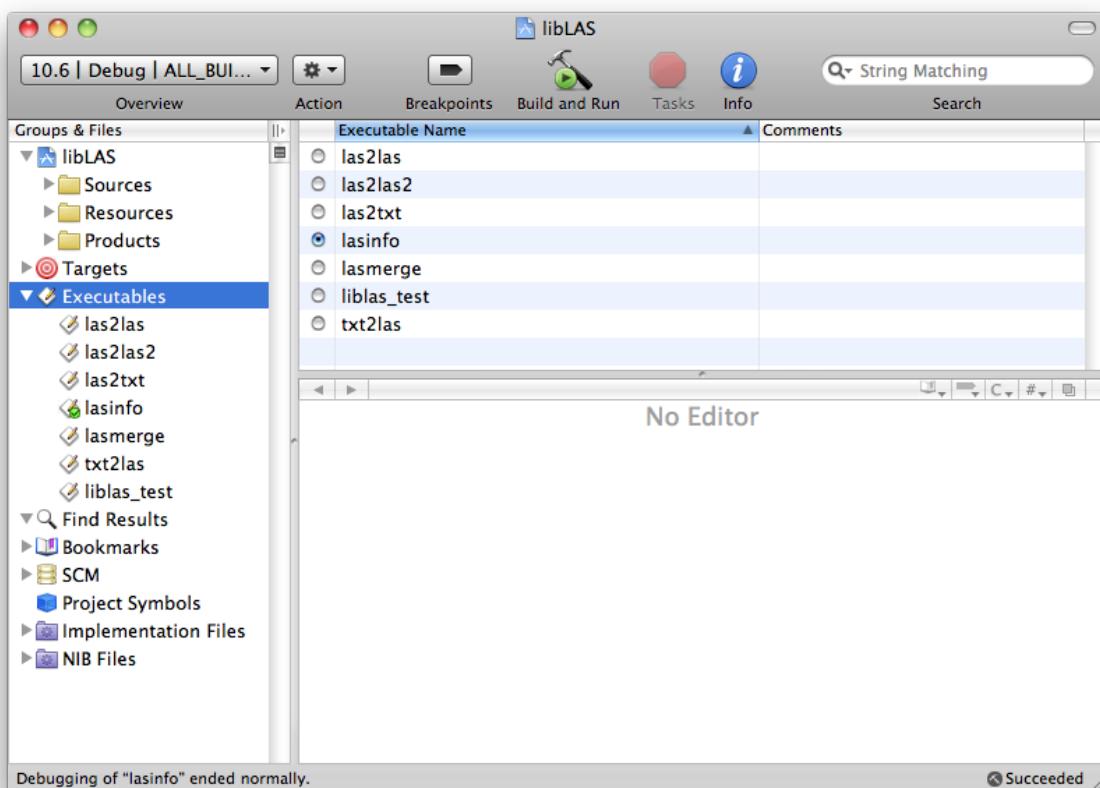
Set default command for XCode

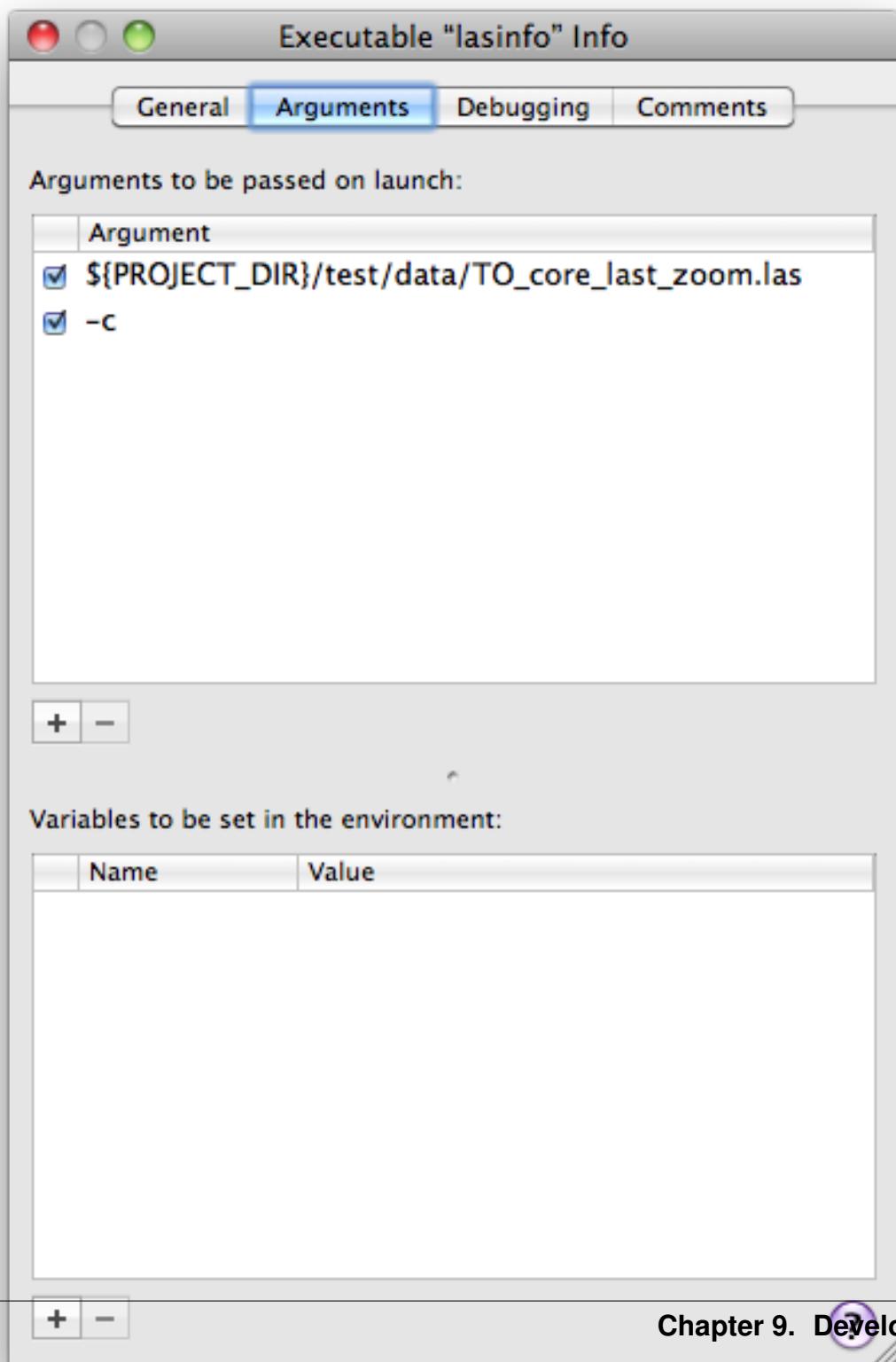
Set the default executable for the project to be `lasinfo` by opening the “Executables” tree, choosing “`lasinfo`,” and clicking the bubble next to the “Executable name” in the right-hand panel.

Set arguments for Testing

Set the arguments for [Testing](#) (page 411) 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 368) above in the “Unix Makefiles” section.







Configure Optional Libraries

As *before* (page 368), use `ccmake . . /` or `cmake-gui . . /` to configure your *Dependencies* (page 390).

Building Under Windows

Author Michael Rosen

Contact unknown at lizardtech dot com

Date 3/19/2012

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 9) 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.

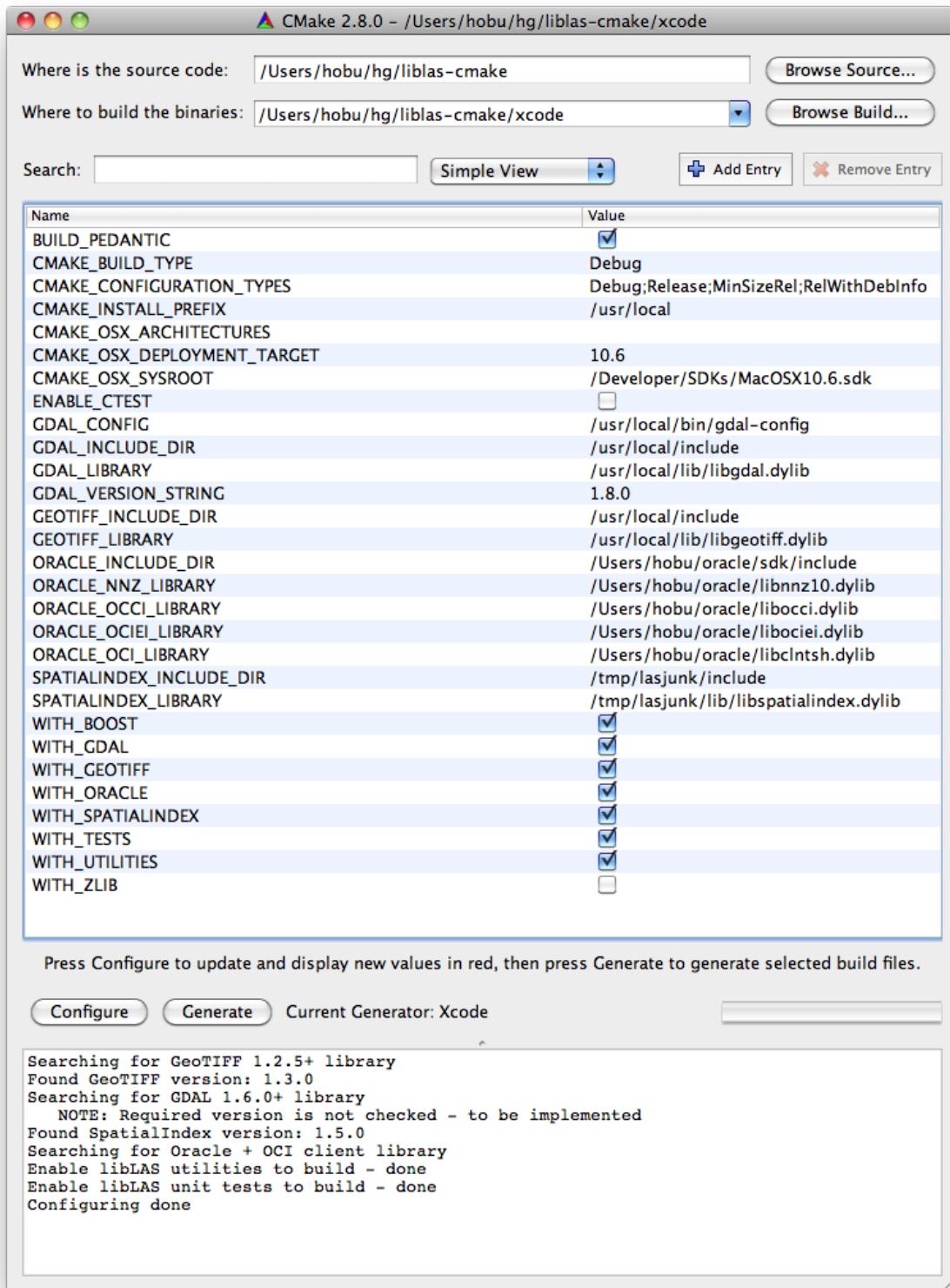
Most Users

If you just want to build PDAL so you can

1. use the utilities or
2. investigate PDAL integration
3. test / fix / extend the library and

you are OK with doing all this with a 32 bit compiler, check the [Prerequisites](#) (page 379) and follow the [Basic Build Steps](#) (page 379) below.

PDAL: Point cloud Data Abstraction Library, 1.4.0



Advanced Users

If your needs go beyond this, then check the *Prerequisites* (page 379) and use the *Advanced Build Steps* (page 388) below as a guide to configuring the library.

Prerequisites

- Ensure you have git. We used version 1.7.3.1.msysgit.0 from <http://code.google.com/p/msysgit>
- Ensure you have a Visual Studio 2010 environment setup. We used VStudio 2010 Premium.
- Ensure you have CMake (<http://www.cmake.org>). We used version 2.8.4 from <http://www.cmake.org>

The steps below assume that these tools are available from your command line. For Git and CMake, that's just adding them to your %PATH%. For VStudio, you need to run *vvars32.bat* (“Microsoft Visual Studio 10.0\Common7\Tools\vvars32.bat”)

Basic Build Steps

Most users can use this procedure to build PDAL on Windows. We satisfy all dependencies using OSGeo4W.

0. If you plan to use LAZ support (compressed LAS), get the LASzip source code and build it (<http://www.laszip.org>). Add the directory with laszip.dll to your %PATH% (or copy the laszip.* files into the bin directory of PDAL itself). Make sure it appears in your PATH before OSGeo4W (as per step 4 below).

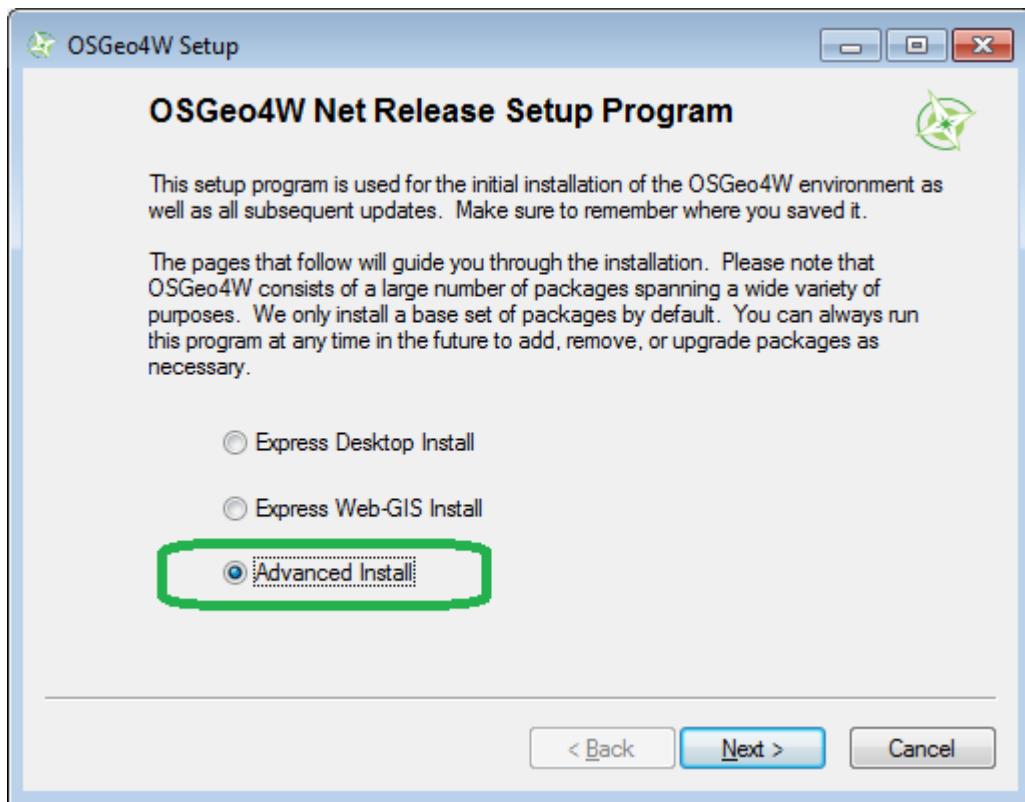
While LASzip is included in the OSGeo4W distribution below (see step 1), the version there is compiled with Visual Studio 2008 and we have some suspicion that it is incompatible with the version of PDAL you are about to build with Visual Studio 2010. Building the LASzip library should be no more complicated than:

```
set G="Visual Studio 10"
set BUILD_TYPE=Debug
cmake -G %G% ^
      -DCMAKE_BUILD_TYPE=%BUILD_TYPE% ^
```

```
-DCMAKE_VERBOSE_MAKEFILE=OFF ^
```

```
.
```

1. Install OSGeo4W (<http://download.osgeo.org/osgeo4w/osgeo4w-setup.exe>) using the “Advanced Install” option and include the following: gdal-dev, laszip, libxml2, iconv.



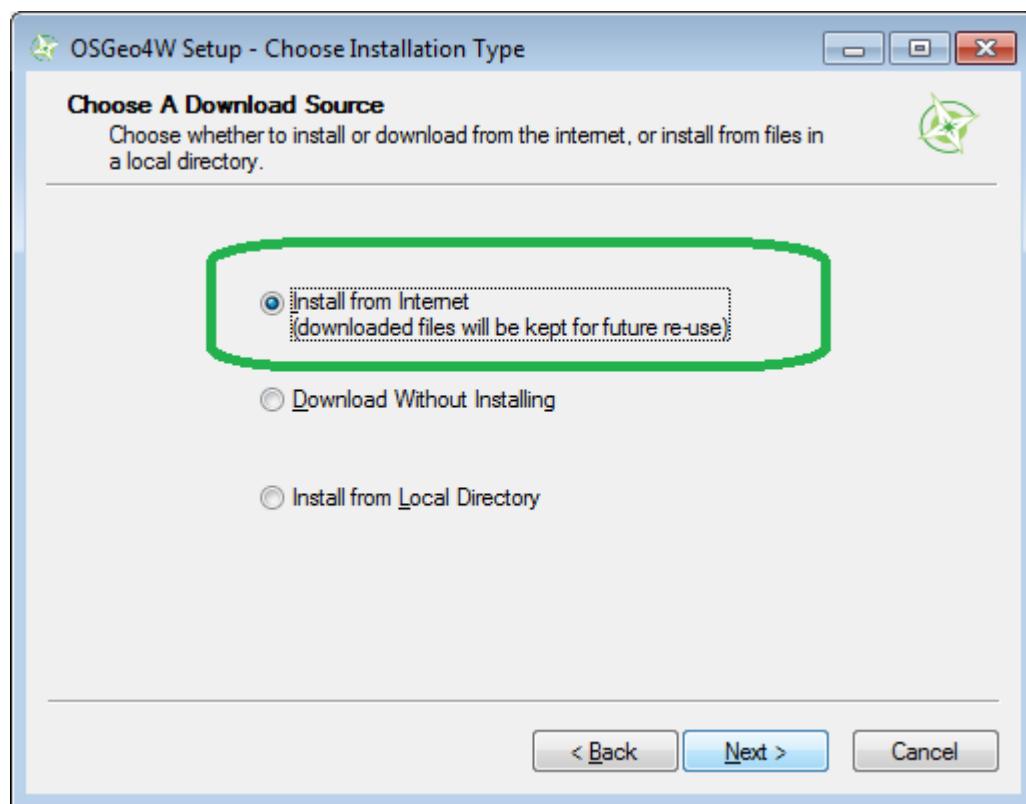
Install from Internet

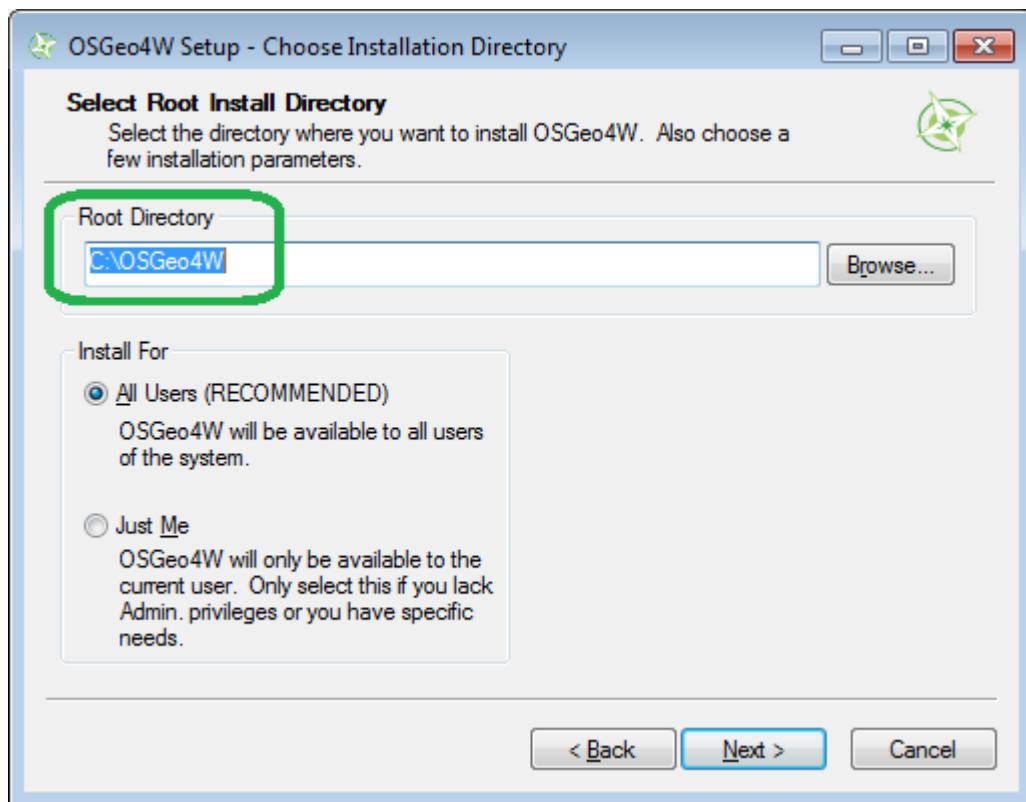
Leave the “Root Directory” for the installation unchanged.

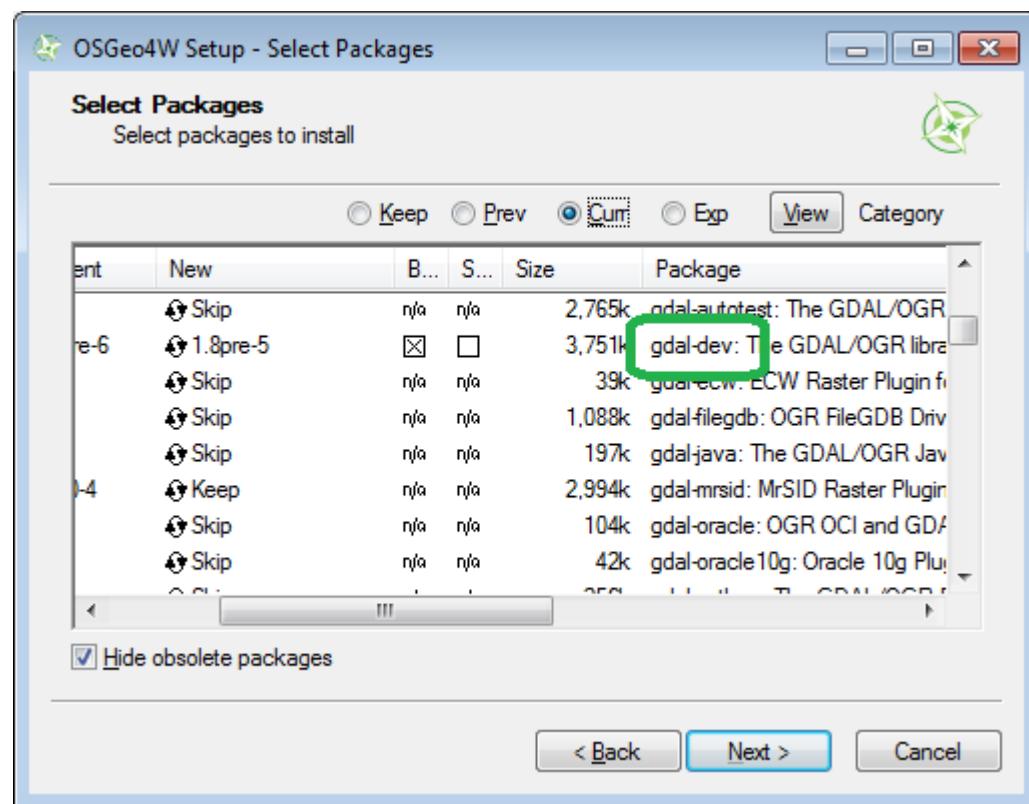
You can leave the “Select Local Package Directory” and “Internet Connection Type” at the defaults.

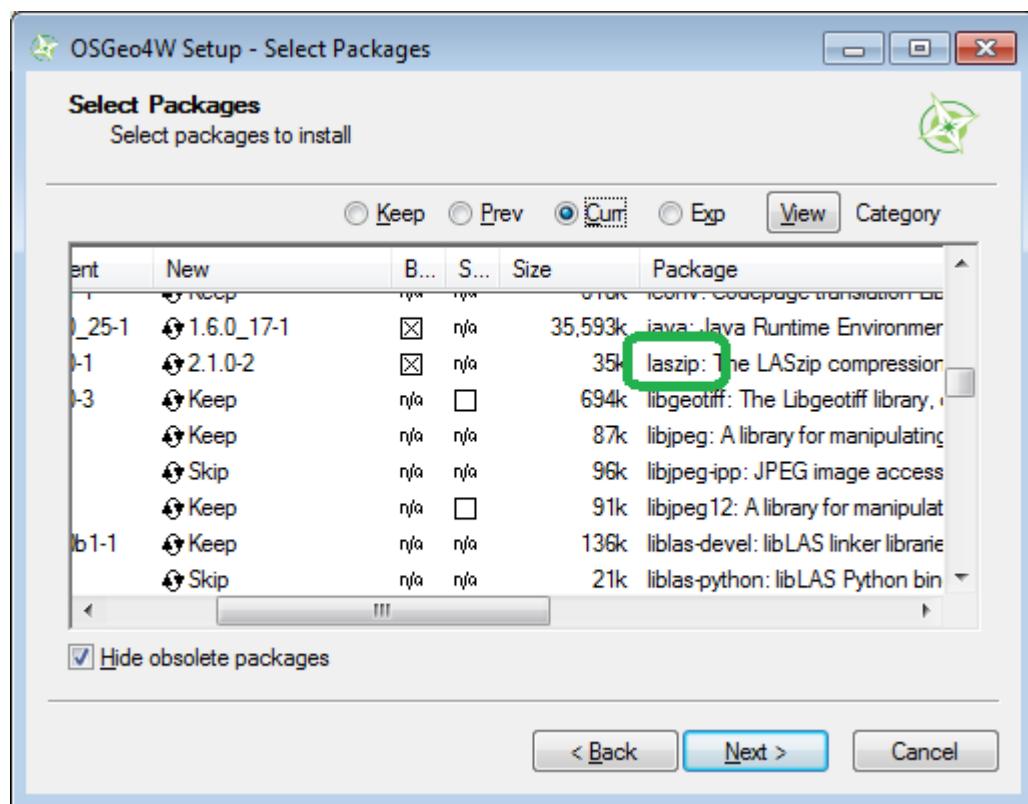
Select “laszip” (this is not required currently – see step 0 above).

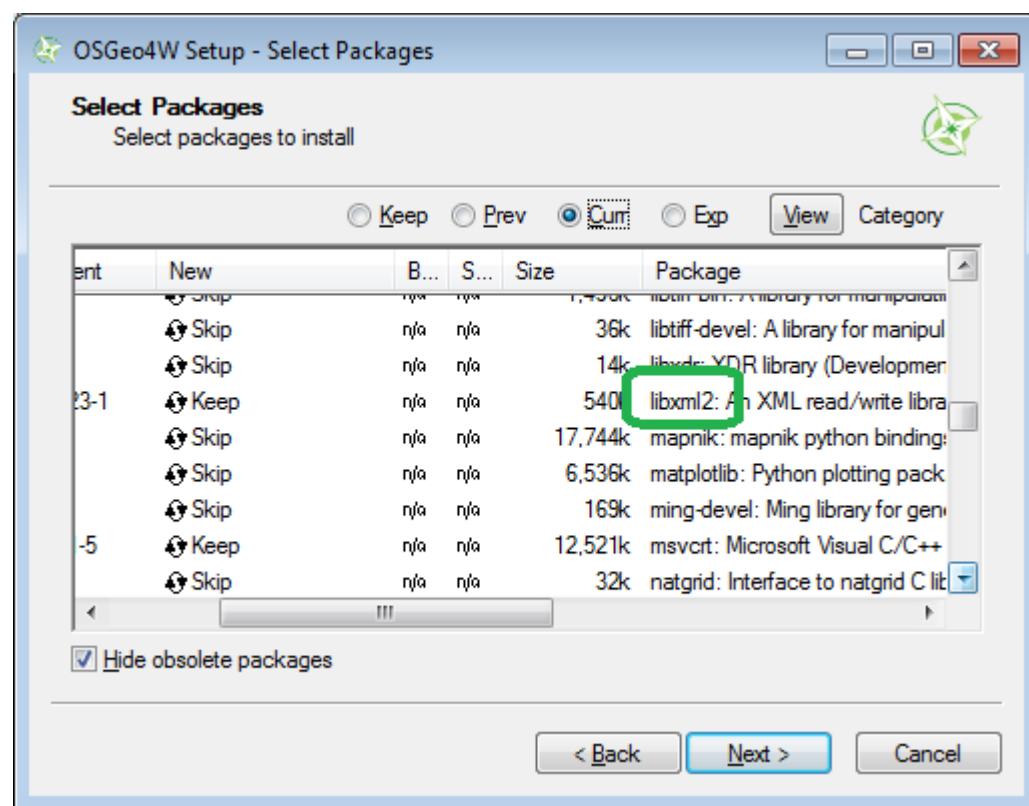
Select “libxml2”.

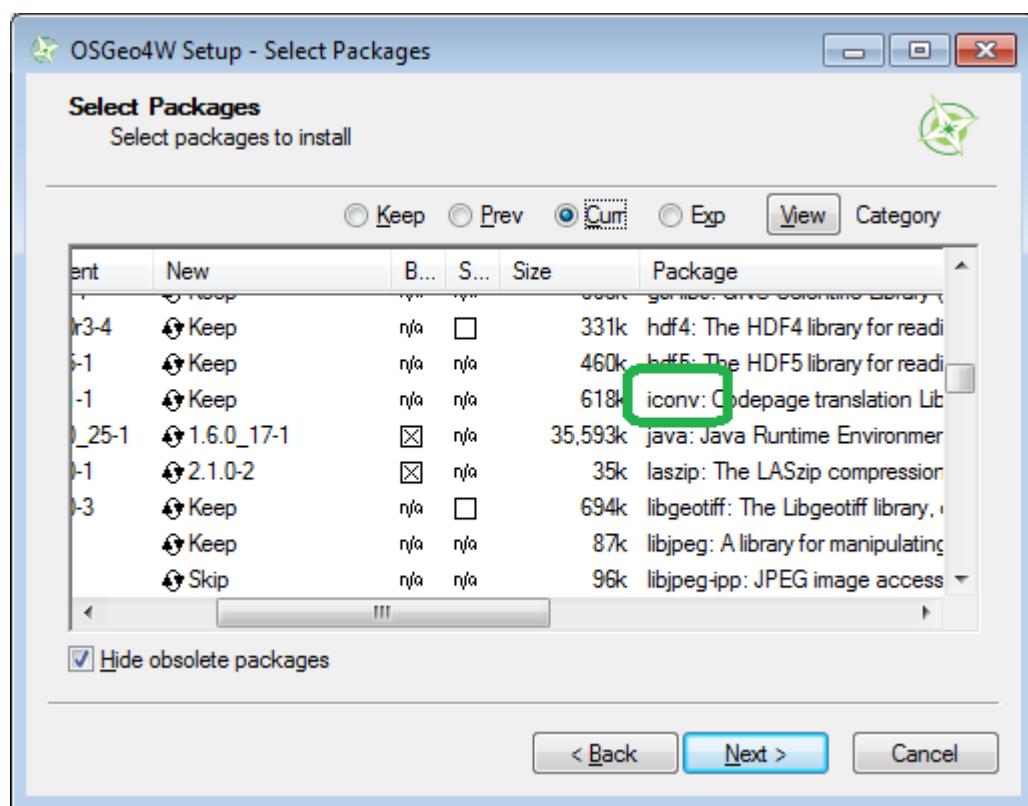






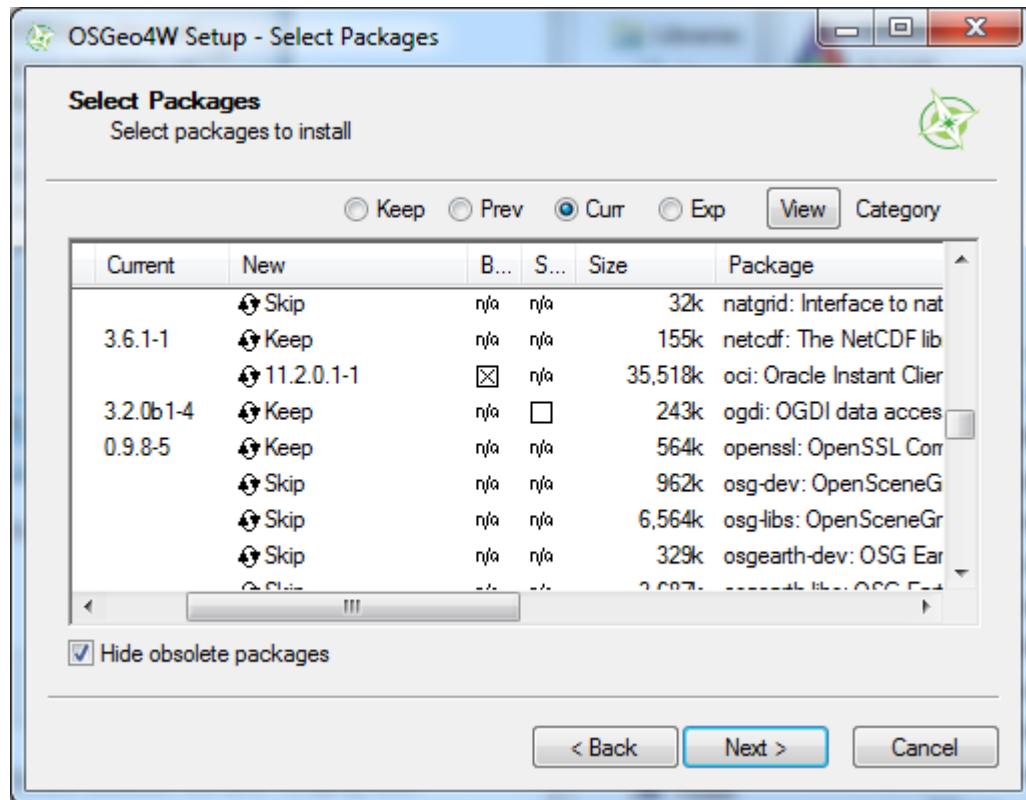






Select “iconv”.

Select “oci”.



There are some other required dependencies (e.g. libtiff, libgeotiff) but they are installed by default.

Select Next to continue on to install the packages.

- Get the source code for PDAL:

```
c:\dev> git clone https://github.com/PDAL/PDAL.git
```

- From the root of your PDAL tree, run “./cmake/examples/mpg-config.bat”. This will create the VStudio solution file and .vcxproj files.

```
c:\dev\PDAL> ./cmake/examples/mpg-config.bat
```

Note: The config.bat file is set up to build PDAL in the “officially supported” configuration – that is, with Oracle, and GDAL, and LASzip, and such. If you followed the previous steps, you should be fine to use this default configuration. You may modify this file if you need to (such as to use a local copy of GDAL or to use NMake instead of Visual Studio); see the [Advanced Build Steps](#) (page 388) below for more instructions.

4. Verify your system environment variables are set properly:
 - PATH should include %OSGeo4W%bin
 - GDAL_DATA should be set to %OSGeo4W%shareepsg_csv
 - PROJ_LIB should be set to %OSGeo4W%shareproj
5. Start Visual Studio and open PDAL.sln. Build the solution (F6).
6. Set pdal_test as the startup project and run it (F5). You should see a console window startup, print something like “Running 158 test cases...” (exact number may vary), and then after a short period print something like “*** No errors detected”. If you do get errors, that means either something is broken on the version of PDAL you checked out OR something is wrong with your installation.
7. PYTHON/PLANG NOTE: If you build WITH_PLANG=ON in Debug mode, the system will try to link against “python27d.lib”. You need to change .../Python27/include/pyconfig.h as follows:
 - **change the line #pragma comment(lib,”python27d.lib”) to refer to python27.lib instead**
 - comment out the line “#define Py_DEBUG”

Advanced Build Steps

Advanced users can use this procedure to customize their PDAL build on Windows. This enables the use of custom-built external libraries to satisfy situations (including x64 support) where using OSGeo4W is inadequate.

1. Acquire and stage the required dependent libraries. PDAL depends on the following external libraries. You’ll need to get them and build them (or perhaps, download prebuilt

binary packages).

- GDAL (get version 1.6 or later from <http://gdal.org>) [optional]
- LASZip (get version 1.0.1 or later from <http://laszip.org>) [optional]
- libxml2 (<http://libxml2.org>) [optional]
- iconv (<http://www.gnu.org/software/libiconv/>) [optional (required by libxml2)]
- oci (optional, <http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html>)
- libtiff (optional)
- libgeotiff (optional)

One option for all dependencies is OSGeo4W (free, win32 installer, no x64 – see “Basic Build Steps”_ above).

2. Having staged the above libs, you need to specify where they are by editing the appropriate lines in the “config.bat” file. Each dependency has a short section of the config.bat file. Check these are specified correctly. For example

```
:: LASZIP
set LASZIP_ENABLED=ON
set LASZIP_LIBRARY=%DEV_DIR%\laszip\bin\Debug\laszip.lib
set LASZIP_INCLUDE_DIR=%DEV_DIR%\laszip\include
```

3. While you’re still in config.bat, configure the “Generator” for CMake.

```
:: Pick your Generator. NMake will pick up architecture_
→(x32, x64) from your environment
rem set GENERATOR="NMake Makefiles"
rem set GENERATOR="Visual Studio 10 Win64"
set GENERATOR="Visual Studio 10"

set BUILD_TYPE=Release
rem set BUILD_TYPE=Debug
```

4. Run “config.bat” in the PDAL directory. This will create the VStudio solution file and .vcxproj files.

```
c:\dev\PDAL> config.bat
```

5. Start Visual Studio and open PDAL.sln (or if you picked “NMake Makefiles”, run NMake).

Testing

Set pdal_test as the default/startup application in Visual Studio Run with debug (F5)

Troubleshooting

- Missing libblas.dll - double check that C : \OSGeo4W\bin is on your system PATH variable
- libtiff.dll errors - double check that other versions of the lib are not on the path. For example, ArcGIS installs a version of libtiff that is not compatible.
- “ERROR 4: Unable to open EPSG support file gcs.csv” or GDAL_DATA variable errors
 - Set GDAL_DATA system variable to C : \OSGeo4W\share\gdal
- PROJ errors - Set PROJ_LIB system variable to C : \OSGeo4W\share\proj

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 89) 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) (<http://www.gdal.org>) via whatever method is convenient. Linux platforms such as [Debian](http://www.debian.org) (<http://www.debian.org>) have [DebianGIS](http://wiki.debian.org/DebianGis) (<http://wiki.debian.org/DebianGis>), Mac OS X has the [KyngChaos](http://www.kyngchaos.com/software/unixport) (<http://www.kyngchaos.com/software/unixport>) software frameworks, and Windows has the [OSGeo4W](http://trac.osgeo.org/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.

Optional Dependencies

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>) is the projection engine that PDAL uses for the *filters.reprojection* (page 152) 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.

libxml2

libxml2 (<http://xmlsoft.org>) is used to serialize PDAL dimension descriptions into XML for the database drivers such as *writers.oci* (page 89), *readers.sqlite* (page 70), or *readers.pgpointcloud* (page 63)

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 89) and *readers.oci* (page 61) for Oracle Point Cloud read/write support.

Warning: [OCI](http://www.oracle.com/technology/tech/oci/index.html) (<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) (<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.

Points2Grid

[Points2Grid](https://github.com/CRREL/points2grid) (<https://github.com/CRREL/points2grid>) is a library with a simple *CMake*-based build system that provides simple, out-of-process interpolation of large point sets using [Boost](http://www.boost.org) (<http://www.boost.org>). It can be obtained via github.com at <https://github.com/CRREL/points2grid>. It is used by [writers.p2g](#) (page 92) to output point cloud interpolation.

Hexer

[Hexer](https://github.com/hobu/hexer) (page 393) 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>. It is used by [filters.hexbin](#) (page 123) 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 86). 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>) 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>) data. It is used by the [*writers.las*](#) (page 82) and [*readers.las*](#) (page 57) to provide compressed LAS support.

laz-perf

In addition to [LASzip](#) (<http://laszip.org>), you can use the alternative [laz-perf](#) (<https://github.com/verma/laz-perf/>) library. [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 89) and [*writers.sqlite*](#) (page 98)

PCL

The Point Cloud Library (PCL) (<http://pointclouds.org>) is used by the [*pcl*](#) (page 26), [*writers.pcd*](#) (page 93), [*readers.pcd*](#) (page 63), and [*filterspclblock*](#) (page 141) 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>)-based OSX builds use PCL 1.7.2, but you may need to switch off [VTK](#) (<http://vtk.org>) support depending on the configuration.

Python installation

Beginning in PDAL 1.0.0, a Python extension to execute pipelines and read its output as a numpy array is available.

To install it need to compile and install *PDAL* (page 365) and then install the PDAL Python extension

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/>)

Coding Conventions

To the extent possible and reasonable, we value consistency of source code formatting, class and variable naming, and so forth.

This Note lists some such conventions that we would like to follow, where it makes sense to do so.

Source Formatting

We use astyle (<http://astyle.sourceforge.net>) as a tool to reformat C++ source code files in a consistent fashion. The file astylerc, at the top of the github repo, contains the default settings we use.

Our conventions are:

- LF endings (unix style), not CRLF (windows style)
- spaces, not tabs
- indent to four (4) spaces (“Four shalt be the number thou shalt count, and the number of the counting shall be four. Three shalt thou not count, neither count thou five...”)
- braces shall be on their own lines, like this:

```
if (p)
{
    foo();
}
```

- copyright header, license, and author(s) on every file
- two spaces between major units, e.g. function bodies

Naming Conventions

- classes should be names using UpperCamelCase
- functions should be in lowerCamelCase
- member variables should be prefixed with “m_”, followed by the name in lowerCamelCase – for example, “m_numberOfPoints”
- there should be only one class per file, and the name of the file should match the class name – that is, class PointData should live in files PointData.hpp and PointData.cpp

Other Conventions

- the use of getter and setter methods is preferred to exposing member variables

- Surround all code with “namespace pdal {...}”; where justifiable, you may introduce a nested namespace.
- Use exceptions for exceptional events that are not going to be handled directly within the context of where the event occurs. Avoid status codes. See `pdal_error.hpp` for a set of pdal-specific exception types you may throw.
- Describe use of “debug” and “verbose” settings.
- Don’t put member function bodies in the class declaration in the header file, unless clearly justified for performance reasons. Use the “inline” keyword in these cases(?).
- Use `const`.
- Don’t put “using” declarations in headers.
- Document all public (and protected) member functions using doxygen markup.

Layout/Organization of Source Tree

- public headers in `./include`
- private headers alongside source files in `src/`
- ...

#include Conventions

- For public headers from the `./include/pdal` directory, use angle brackets: `#include <pdal/Stage.h>`
- For private headers (from somewhere in `./src`), use quotes: `#include "support.hpp"`
- `#include` lines should be grouped and arranged in this order: C++/std headers, 3rd-party headers (e.g. gdal), pdal headers, local headers. The pdal headers may be further grouped by subdirectory, e.g. drivers/libblas, filters, etc.
- Exception to the above: source files (`.cpp`) should `#include` their corresponding `.hpp` file first. This assures that the header is including all the files it needs to.
- Don’t `#include` a file where a simple forward declaration will do. (Note: this only applies to pdal files; don’t forward declare from system or 3rd party headers.)

- Don't include a file unless it actually is required to compile the source unit.
- Don't use manual include guards. All reasonable compilers support the once pragma:

```
#pragma once
```

Contributors

Numerous organizations, companies, and individuals have contributed time, money, and code to build PDAL up into a highly capable software package. Without these contributions, PDAL would not progress as quickly, and its quality wouldn't be as high. The development team is proud of the software, and it collectively represents years of experiences doing point cloud data management. We hope you'll find it useful too.

This page is to recognize these contributors and their contributions. Thanks.

Engineering Contributors

(<http://hobu.co>) **Hobu** (<http://hobu.co>) is the primary company behind the design, testing, development, and distribution of PDAL. Two Hobu team members primarily interact with PDAL. **Howard Butler** (<https://github.com/hobu>) founded the project, and he provides project leadership and software development. **Andrew Bell** (<https://github.com/abellgithub>) has contributed design, refactoring, and new feature development of PDAL over the past couple of years. (<http://radiantblue.com>) **Michael Gerlek** (<http://github.com/mpgerlek>) from **RadiantBlue** (<http://radiantblue.com/>) helped bootstrap PDAL by providing its first design, basic primitive objects, and first stage implementations.

Bradley Chambers (<https://github.com/chambbj>) from **RadiantBlue** (<http://radiantblue.com/>) has contributed numerous features and capabilities to the PDAL project, including *Poisson sampling* (page 219) and *Progressive Morphological Filters* (page 197). He is also a prolific *Tutorials* (page 165) writer.

Funding Contributors



(<http://www.erdc.usace.army.mil/Locations/CRREL.aspx>) The US Army Corps of Engineers Remote Sensing / GIS Center of Expertise at [CRREL](http://www.erdc.usace.army.mil/Locations/CRREL.aspx) (<http://www.erdc.usace.army.mil/Locations/CRREL.aspx>) sponsors development of PDAL for its use in point cloud data management systems. [CRREL](#) (<http://www.erdc.usace.army.mil/Locations/CRREL.aspx>)’s [GRiD](#) (<http://lidar.io/about.html>) project manages LiDAR and point cloud data for a multitude of U.S. Army Corps missions. Find out more about GRiD in this [LiDAR Magazine article](#)



(<http://www.lidarmag.com/content/view/11343/198/>).



(<http://www.nsf.gov>) (<http://www.uh.edu>) **NSF**
(<http://www.nsf.gov>), in collaboration with **Dr. Craig Glennie**
(<http://www.cive.uh.edu/faculty/glennie>) at the **University of Houston** (<http://www.uh.edu>)
supports PDAL with funding support to develop and enhance statistical methods,
transformation operations, tutorial and example development, and **PCL** (<http://pointclouds.org>)
integration.

Docs

Requirements

To build the PDAL documentation yourself, you need to install the following items:

- [Sphinx](http://sphinx-doc.org/) (<http://sphinx-doc.org/>)
- [Breathe](https://github.com/michaeljones/breathe) (<https://github.com/michaeljones/breathe>)
- [Doxygen](http://www.stack.nl/~dimitri/doxygen/) (<http://www.stack.nl/~dimitri/doxygen/>)
- [Latex](https://en.wikipedia.org/wiki/LaTeX) (<https://en.wikipedia.org/wiki/LaTeX>)
- [dvipng](https://en.wikipedia.org/wiki/Dvipng) (<https://en.wikipedia.org/wiki/Dvipng>)

Note: For the website, PDAL builds the documentation using [Travis](#) (page 420) and [Install Docker](#) (page 9). You can get a listing of the exact Ubuntu requirements in the [Dockerfile](#) (<https://github.com/PDAL/PDAL/tree/master/scripts/docker/docbuild/Dockerfile>). See [Continuous Integration](#) (page 420) for more detail.

Sphinx (<http://sphinx-doc.org/>) and Breathe (<https://github.com/michaeljones/breathe>)

Python dependencies should be installed from [PyPI](https://pypi.python.org/pypi) (<https://pypi.python.org/pypi>) with `pip` or `easy_install`.

```
(sudo) pip install sphinx sphinxconfig-bibtex breathe
```

Note: If you are installing these packages to a system-wide directory, you may need the `sudo` in front of the `pip`, though it might be better that instead you use [virtual environments](https://pypi.python.org/pypi/virtualenv) (<https://pypi.python.org/pypi/virtualenv>) instead of installing the packages system-wide.

Doxygen

The PDAL documentation also depends on Doxygen (<http://www.stack.nl/~dimitri/doxygen/>), which can be installed from source or from binaries from the [doxygen website](http://www.stack.nl/~dimitri/doxygen/download.html) (<http://www.stack.nl/~dimitri/doxygen/download.html>). If you are on Max OS X and use [homebrew](http://mxcl.github.io/homebrew/) (<http://mxcl.github.io/homebrew/>), you can install doxygen with a simple `brew install doxygen`.

Latex

[Latex](https://en.wikipedia.org/wiki/LaTeX) (<https://en.wikipedia.org/wiki/LaTeX>) and [pdflatex](https://www.tug.org/applications/pdflatex/) (<https://www.tug.org/applications/pdflatex/>) are used to generate the companion PDF of the website.

dvipng

For math output, we depend on [dvipng](https://en.wikipedia.org/wiki/Dvipng) (<https://en.wikipedia.org/wiki/Dvipng>) to turn [Latex](https://en.wikipedia.org/wiki/LaTeX) (<https://en.wikipedia.org/wiki/LaTeX>) output into math PNGs.

Generation

Once you have installed all the doc dependencies, you can then build the documentation itself. The `doc` / directory in the PDAL source tree contains a Makefile which can be used to build all documentation. For a list of the output formats supported by Sphinx, simply type `make`. For example, to build html documentation:

```
cd doc  
make doxygen html
```

The html docs will be placed in `doc/build/html/`. The `make doxygen` is necessary to re-generate the API documentation from the source code using [Breathe](#) (<https://github.com/michaeljones/breathe>) and [Sphinx](#) (<http://sphinx-doc.org/>).

Note: For a full build of the [C++ API](#) (page 421) documentation, you need to make `doxygen` to have it build its XML output which is consumed by [Breathe](#) (<https://github.com/michaeljones/breathe>) before `make html` can be issued.

Website

The <http://pdal.io> website is regenerated from the `*-maintenance` branch using [Travis](#) (page 420). It will be committed by the `PDAL-docs` [GitHub](#) (<http://github.com/PDAL/PDAL>) user and pushed to the <https://github.com/PDAL/pdal.github.io> repository. The website is then served via [GitHub Pages](#) (<https://pages.github.com/>).

Note: The website is regenerated and pushed only on the `after_success` [Travis](#) (page 420) call. If the tests aren't passing, the website won't be updated.

Building With Docker

A [Install Docker](#) (page 9) image, `pdal/docs` contains the full compliment of requirements, and it is used by PDAL's [Travis](#) (page 420) continuous integration to build and commit new versions of the website. You can easily build the docs using Docker by issuing the following command:

```
docker run -v /path/to/pdal/root/tree:/data -w /data/doc pdal/docs  
→make html
```

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.

Metadata

Metadata is an important component of any data processing story. PDAL attempts to allow users to operate with metadata in a relatively free-form way, with its main Metadata utility,

`pdal::Metadata.`

The basic structure of a `pdal::Metadata` instance is the following tree structure:

```
{  
    "type": "blank",  
    "value": "",  
    "name": "name",  
    "metadata": {}  
}
```

Note: Metadata instances can contain other Metadata instances by adding them with the `pdal::Metadata::addMetadata()` method. They will be added to the *metadata* sub-tree of the internal `property_tree`.

Metadata Types

`pdal::Metadata` instances require that any classes that are added to them be copy-constructable and have an `ostream<<` operator for them. While these constraints mean they are all serializable to strings, `pdal::Metadata` also keeps an explicit type variable, `type` for each instance. This property allows us to say something extra about the Metadata entry, and allows them to go in and out of `pdal::Stage` and `pdal::PointView` with type fidelity.

The metadata `type` variable roughly maps to the [XSD type names](#) (<http://infohost.nmt.edu/tcc/help/pubs/rnc/xsd.html>). The following types are valid for current PDAL versions, though more may be added.

Table 9.1: PDAL `pdal::Metadata` types

double	float	integer
nonNegativeInteger	boolean	string
base64Binary	uuid	bounds
spatialreference	blank	

Warning: Explicitly-sized types are not supported. Assume that *integer* or *nonNegativeInteger* map to the typical 4-byte signed and unsigned types. You might be required to adjust the value based on an explicit interpretation and cast it into these larger types.

JSON (<http://www.json.org/>) representation

A more interesting metadata tree might come from the `pdal::drivers::las::Reader`. Interesting things to note include

```
{
    "name": "readers.las",
    "type": "blank",
    "value": "",
    "metadata":
    {
        "compressed":
        {
            "name": "compressed",
            "description": "true if this LAS file is compressed",
            "type": "boolean",
            "value": "false"
        },
        "dataformatid":
        {
            "name": "dataformatid",
            "description": "The Point Format ID as specified in the
                           ↪LAS specification",
            "type": "nonNegativeInteger",
            "value": "3"
        },
        ...
        "project_id":
        {
            "name": "project_id",
            "description": "Project ID (GUID data): The four fields
                           ↪that comprise a complete Globally Unique Identifier (GUID) are now
                           ↪reserved for use as a Project Identifier (Project ID). The field
                           ↪remains optional. The time of assignment of the Project ID is at
                           ↪the discretion of processing software. The Project ID should be
                           ↪the same for all files that are associated with a unique project."
        }
    }
}
```

9.1. Development 405
 ↪By assigning a Project ID and using a File Source ID (defined
 ↪above) every file within a project and every point within a file
 ↪can be uniquely identified, globally.",

```
        "type": "uuid",
        "value": "00000000-0000-0000-0000-000000000000"
    },
    "system_id":
    {
        "name": "system_id",
        "description": "",
        "type": "string",
        "value": "HOBUSYSTEMID"
    },
    ...
    "vlr_0":
    {
        "name": "vlr_0",
        "description": "A Polygon WKT entry",
        "type": "base64Binary",
        "value": "UE9MWUdPTigoNiAxNSwgMTAgMTAsIDIwIDEwLCAYNSAxNSwgMjUgMzUsIDE5IDQwLCAxMSA0MCwgNiA0
        "metadata":
        {
            "reserved":
            {
                "name": "reserved",
                "description": "Two bytes of padded, unused space. Some softwares expect the values of these bytes to be 0xAABB as specified in the 1.0 version of the LAS specification",
                "type": "nonNegativeInteger",
                "value": "43707"
            },
            "user_id":
            {
                "name": "user_id",
                "description": "The User ID field is ASCII character data that identifies the user which created the variable length record. It is possible to have many Variable Length Records from different sources with different User IDs. If the character data is less than 16 characters, the remaining data must be null. The User ID must be registered with the LAS specification managing body. The management of these User IDs ensures that no two individuals accidentally use the same User ID. The specification will initially use two IDs: one for globally specified records (LASF_Spec), and another for projection types (LASF_Projection). Keys may be requested at http://www.asprs.org/lasform/keyform.html."
            }
        }
    }
}
```

```
        "type": "string",
        "value": "hobu"
    },
    "record_id":
    {
        "name": "record_id",
        "description": "The Record ID is dependent upon the User ID. There can be 0 to 65535 Record IDs for every User ID. The LAS specification manages its own Record IDs (User IDs owned by the specification), otherwise Record IDs will be managed by the owner of the given User ID. Thus each User ID is allowed to assign 0 to 65535 Record IDs in any manner they desire. Publicizing the meaning of a given Record ID is left to the owner of the given User ID. Unknown User ID\Record ID combinations should be ignored."
    },
    "type": "nonNegativeInteger",
    "value": "1234"
},
"description":
{
    "name": "description",
    "description": "",
    "type": "string",
    "value": "A Polygon WKT entry"
}
}
},
...
}
```

Pipeline XML representation

The *Pipeline* (page 37) representation of the `pdal::Metadata` is a little bit flatter...

```
<?xml version="1.0" encoding="utf-8"?>
<Reader type="readers.las">
    <Option name="debug">false</Option>
    <Option name="filename">test/data/interesting.las</Option>
```

```
<Option name="verbose">0</Option>
<Metadata name="writers.las" type="blank">
    <Metadata name="compressed" type="boolean">false</Metadata>
    <Metadata name="dataformatid" type="nonNegativeInteger">3</
    <Metadata>
        <Metadata name="version_major" type="nonNegativeInteger">1</
    <Metadata>
        <Metadata name="version_minor" type="nonNegativeInteger">2</
    <Metadata>
        <Metadata name="filesource_id" type="nonNegativeInteger">0</
    <Metadata>
        <Metadata name="reserved" type="nonNegativeInteger">0</Metadata>
        <Metadata name="project_id" type="uuid">00000000-0000-0000-
        <000000000000</Metadata>
        <Metadata name="system_id" type="string">HOBU-SYSTEMID</Metadata>
        <Metadata name="software_id" type="string">HOBU-GENERATING</
    <Metadata>
        <Metadata name="creation_doy" type="nonNegativeInteger">145</
    <Metadata>
        <Metadata name="creation_year" type="nonNegativeInteger">2012</
    <Metadata>
        <Metadata name="header_size" type="nonNegativeInteger">227</
    <Metadata>
        <Metadata name="dataoffset" type="nonNegativeInteger">1488</
    <Metadata>
        <Metadata name="scale_x" type="double">0.01</Metadata>
        <Metadata name="scale_y" type="double">0.01</Metadata>
        <Metadata name="scale_z" type="double">0.01</Metadata>
        <Metadata name="offset_x" type="double">-0</Metadata>
        <Metadata name="offset_y" type="double">-0</Metadata>
        <Metadata name="offset_z" type="double">-0</Metadata>
        <Metadata name="minx" type="double">635619.85</Metadata>
        <Metadata name="miny" type="double">848899.7000000001</Metadata>
        <Metadata name="minz" type="double">406.59</Metadata>
        <Metadata name="maxx" type="double">638982.55</Metadata>
        <Metadata name="maxy" type="double">853535.4300000001</Metadata>
        <Metadata name="maxz" type="double">586.38</Metadata>
        <Metadata name="count" type="nonNegativeInteger">1065</Metadata>
        <Metadata name="vlr_0" type="base64Binary">
            <UE9MWUdPTigoNiAxNSwgMTAgMTAsIDIwIDEwLCAyNSAxNSwgMjUgMzUsIDE5IDQwLCAxMSA0MCwgNiAy>
        <UE9MWUdPTigoNiAxNSwgMTAgMTAsIDIwIDEwLCAyNSAxNSwgMjUgMzUsIDE5IDQwLCAxMSA0MCwgNiAy>
    <Metadata>
```

```

        <Metadata name="reserved" type="nonNegativeInteger">43707</
    ↵Metadata>
        <Metadata name="user_id" type="string">hobu</Metadata>
        <Metadata name="record_id" type="nonNegativeInteger">1234</
    ↵Metadata>
        <Metadata name="description" type="string">A Polygon WKT_U
    ↵entry</Metadata>
    </Metadata>
    <Metadata name="vlr_1" type="base64Binary">
    ↵AQABAAAAFQAABAAAAQABAEEAAABAAEAAgSxhywAAAACAAAAQD/
    ↵fwEIsYdqACwAAggAAEA/38GCAAAQCOIwgIAAABAP9/
    ↵CQiwhwEABgALCLCHAQAHAA0IsIcBAAgAAAwAAAEE/38CDAAAAQD/
    ↵fwMMAAABAAgABAwAAAEEAKiMGDLCHAQACAAcMsIcBAAMADAywhwEAAQANDLCHAQAAAA4MsIcBAAQADwy
        <Metadata name="reserved" type="nonNegativeInteger">43707</
    ↵Metadata>
        <Metadata name="user_id" type="string">LASF_Projection</
    ↵Metadata>
        <Metadata name="record_id" type="nonNegativeInteger">34735</
    ↵Metadata>
        <Metadata name="description" type="string">GeoTIFF_U
    ↵GeoKeyDirectoryTag</Metadata>
    </Metadata>
    <Metadata name="vlr_2" type="base64Binary">
    ↵AAAAAAADgREAAAAAAACBewAAAAAAAgEVAAAAAAADARKD/////
    ↵2kYQQAAAAAAAAAAAAAQKZUWEGo+euUHaRyQAAAAAAAAAAA
        <Metadata name="reserved" type="nonNegativeInteger">43707</
    ↵Metadata>
        <Metadata name="user_id" type="string">LASF_Projection</
    ↵Metadata>
        <Metadata name="record_id" type="nonNegativeInteger">34736</
    ↵Metadata>
        <Metadata name="description" type="string">GeoTIFF_U
    ↵GeoDoubleParamsTag</Metadata>
    </Metadata>
    <Metadata name="vlr_3" type="base64Binary">
    ↵TkFEXzE5ODNfT3J1Z29uX1N0YXR1d2lkZV9MYW1iZXJ0X0ZlZXRFsW50bHxHQ1MgTmFtZSA9IEdDU19
        <Metadata name="reserved" type="nonNegativeInteger">43707</
    ↵Metadata>
        <Metadata name="user_id" type="string">LASF_Projection</
    ↵Metadata>

```

```
<Metadata name="record_id" type="nonNegativeInteger">34737</
<Metadata name="description" type="string">GeoTIFF<
<GeoAsciiParamsTag></Metadata>
</Metadata>
<Metadata name="vlr_4" type="base64Binary">
<UFJPSSkNTWyJOQURfMTk4M19PcmVnb25fU3RhdGV3aWRlX0xhbWJlcRfRmVldF9JbnRsIixHRU9HQ1Nb
<Metadata name="reserved" type="nonNegativeInteger">43707</
<Metadata name="user_id" type="string">liblas</Metadata>
<Metadata name="record_id" type="nonNegativeInteger">2112</
<Metadata name="description" type="string">OGR variant of<
<OpenGIS WKT SRS></Metadata>
</Metadata>
</Metadata>
</Reader>
```

Project Goals

1. PDAL is a library which provides APIs for reading, writing, and processing geospatial point cloud data of various formats. Additionally, some command line tools are provided. As [GDAL](http://gdal.org) (<http://gdal.org>) is to 2D pixels, [OGR](http://gdal.org/ogr/) (<http://gdal.org/ogr/>) is to geospatial vector geometry, PDAL is to multidimensional points.
2. From a market perspective, PDAL is “version 2” of libLAS. The actual code base will be different, however, and the APIs will not be compatible.
3. The PDAL implementation has high performance, yet the API remains flexible. We recognize that these two goals will conflict at times and will weigh the tradeoffs pragmatically.
4. The architecture of a PDAL-based workflow will be a pipeline of connected stages, each stage being either a data source (such as a file reader), a filter (such as a point thinner), or data sink (such as a file writer).
5. The PDAL library will be written in C++. PDAL will support multiple platforms, specifically Windows, Linux, and Mac.

6. PDAL is open source and is released under a BSD license. See [License](#) (page 426) for more information.

Testing

Unit Tests

A unit test framework is provided, with the goal that all (nontrivial) classes will have unit tests. At the very least, each new class should have a corresponding unit test file stubbed in, even if there aren't any tests yet.

- Our unit tests also include testing of the command line [Applications](#) (page 17) and (known) plugins.
- We use the [Google C++ Test Framework](#) (<https://code.google.com/p/googletest/>), but a local copy of it is embedded in the PDAL source tree, and you don't have to have it available as a dependency.
- Unit tests for features that are configuration-dependent, e.g. laszip compression, should be put under the same `#ifdef` guards as the classes being tested.
- The Support class, in the `./test/unit` directory, provides some functions for comparing files, etc, that are useful in writing test cases.
- Unit tests should not be long-running.

Running the Tests

To run all unit tests, issue the following command from your build directory:

```
$ ctest
```

`make test` or `ninja test` should still work as well.

Depending on the which optional components you've chose to build, your output should resemble the following:

```
Test project /Users/hobu/dev/git/pdal
  Start 1: pdal_bounds_test
  1/61 Test #1: pdal_bounds_test ..... Passed 0.
→ 02 sec
```

```
    Start  2: pdal_config_test
2/61 Test #2: pdal_config_test ..... Passed  0.
→02 sec
    Start  3: pdal_file_utils_test
3/61 Test #3: pdal_file_utils_test ..... Passed  0.
→02 sec
    Start  4: pdal_georeference_test
4/61 Test #4: pdal_georeference_test ..... Passed  0.
→02 sec
    Start  5: pdal_kdindex_test
5/61 Test #5: pdal_kdindex_test ..... Passed  0.
→03 sec
    Start  6: pdal_log_test
6/61 Test #6: pdal_log_test ..... Passed  0.
→03 sec
    Start  7: pdal_metadata_test
7/61 Test #7: pdal_metadata_test ..... Passed  0.
→02 sec
    Start  8: pdal_options_test
8/61 Test #8: pdal_options_test ..... Passed  0.
→02 sec
    Start  9: pdal_pdalutils_test
9/61 Test #9: pdal_pdalutils_test ..... Passed  0.
→02 sec
    Start 10: pdal_pipeline_manager_test
10/61 Test #10: pdal_pipeline_manager_test ..... Passed  0.
→03 sec
    Start 11: pdal_point_view_test
11/61 Test #11: pdal_point_view_test ..... Passed  2.
→03 sec
    Start 12: pdal_point_table_test
12/61 Test #12: pdal_point_table_test ..... Passed  0.
→03 sec
    Start 13: pdal_spatial_reference_test
13/61 Test #13: pdal_spatial_reference_test ..... Passed  0.
→07 sec
    Start 14: pdal_support_test
14/61 Test #14: pdal_support_test ..... Passed  0.
→02 sec
    Start 15: pdal_user_callback_test
```

```

15/61 Test #15: pdal_user_callback_test ..... Passed 0.
  ↵02 sec
    Start 16: pdal_utils_test
16/61 Test #16: pdal_utils_test ..... Passed 0.
  ↵02 sec
    Start 17: pdal_lazperf_test
17/61 Test #17: pdal_lazperf_test ..... Passed 0.
  ↵04 sec
    Start 18: pdal_io_bpf_test
18/61 Test #18: pdal_io_bpf_test ..... Passed 0.
  ↵20 sec
    Start 19: pdal_io_buffer_test
19/61 Test #19: pdal_io_buffer_test ..... Passed 0.
  ↵02 sec
    Start 20: pdal_io_faux_test
20/61 Test #20: pdal_io_faux_test ..... Passed 0.
  ↵04 sec
    Start 21: pdal_io_ilvis2_test
21/61 Test #21: pdal_io_ilvis2_test ..... Passed 0.
  ↵06 sec
    Start 22: pdal_io_las_reader_test
22/61 Test #22: pdal_io_las_reader_test ..... Passed 0.
  ↵49 sec
    Start 23: pdal_io_las_writer_test
23/61 Test #23: pdal_io_las_writer_test ..... Passed 2.
  ↵27 sec
    Start 24: pdal_io_optech_test
24/61 Test #24: pdal_io_optech_test ..... Passed 0.
  ↵03 sec
    Start 25: pdal_io_ply_reader_test
25/61 Test #25: pdal_io_ply_reader_test ..... Passed 0.
  ↵03 sec
    Start 26: pdal_io_ply_writer_test
26/61 Test #26: pdal_io_ply_writer_test ..... Passed 0.
  ↵02 sec
    Start 27: pdal_io_qfit_test
27/61 Test #27: pdal_io_qfit_test ..... Passed 0.
  ↵03 sec
    Start 28: pdal_io_sbet_reader_test
28/61 Test #28: pdal_io_sbet_reader_test ..... Passed 0.
  ↵04 sec

```

```
    Start 29: pdal_io_sbet_writer_test
29/61 Test #29: pdal_io_sbet_writer_test ..... Passed 0.
→03 sec
    Start 30: pdal_io_terrassolid_test
30/61 Test #30: pdal_io_terrassolid_test ..... Passed 0.
→03 sec
    Start 31: pdal_filters_chipper_test
31/61 Test #31: pdal_filters_chipper_test ..... Passed 0.
→03 sec
    Start 32: pdal_filters_colorization_test
32/61 Test #32: pdal_filters_colorization_test ..... Passed 11.
→40 sec
    Start 33: pdal_filters_crop_test
33/61 Test #33: pdal_filters_crop_test ..... Passed 0.
→04 sec
    Start 34: pdal_filters_decimation_test
34/61 Test #34: pdal_filters_decimation_test ..... Passed 0.
→02 sec
    Start 35: pdal_filters_divider_test
35/61 Test #35: pdal_filters_divider_test ..... Passed 0.
→03 sec
    Start 36: pdal_filters_ferry_test
36/61 Test #36: pdal_filters_ferry_test ..... Passed 0.
→04 sec
    Start 37: pdal_filters_merge_test
37/61 Test #37: pdal_filters_merge_test ..... Passed 0.
→03 sec
    Start 38: pdal_filters_reprojection_test
38/61 Test #38: pdal_filters_reprojection_test ..... Passed 0.
→03 sec
    Start 39: pdal_filters_range_test
39/61 Test #39: pdal_filters_range_test ..... Passed 0.
→05 sec
    Start 40: pdal_filters_randomize_test
40/61 Test #40: pdal_filters_randomize_test ..... Passed 0.
→02 sec
    Start 41: pdal_filters_sort_test
41/61 Test #41: pdal_filters_sort_test ..... Passed 0.
→39 sec
    Start 42: pdal_filters_splitter_test
```

```

42/61 Test #42: pdal_filters_splitter_test ..... Passed 0.
  ↵03 sec
    Start 43: pdal_filters_stats_test
43/61 Test #43: pdal_filters_stats_test ..... Passed 0.
  ↵03 sec
    Start 44: pdal_filters_transformation_test
44/61 Test #44: pdal_filters_transformation_test ... Passed 0.
  ↵03 sec
    Start 45: pdal_merge_test
45/61 Test #45: pdal_merge_test ..... Passed 0.
  ↵07 sec
    Start 46: pc2pc_test
46/61 Test #46: pc2pc_test ..... Passed 0.
  ↵15 sec
    Start 47: xml_schema_test
47/61 Test #47: xml_schema_test ..... Passed 0.
  ↵02 sec
    Start 48: pdal_filters_attribute_test
48/61 Test #48: pdal_filters_attribute_test ..... Passed 0.
  ↵09 sec
    Start 49: pdal_plugins_cpd_kernel_test
49/61 Test #49: pdal_plugins_cpd_kernel_test ..... ***Exception:_
  ↵Other 0.08 sec
    Start 50: hexbintest
50/61 Test #50: hexbintest ..... Passed 0.
  ↵03 sec
    Start 51: icetest
51/61 Test #51: icetest ..... Passed 0.
  ↵04 sec
    Start 52: mrsidtest
52/61 Test #52: mrsidtest ..... Passed 0.
  ↵06 sec
    Start 53: pdal_io_nitf_writer_test
53/61 Test #53: pdal_io_nitf_writer_test ..... Passed 0.
  ↵08 sec
    Start 54: pdal_io_nitf_reader_test
54/61 Test #54: pdal_io_nitf_reader_test ..... Passed 0.
  ↵04 sec
    Start 55: ocitest
55/61 Test #55: ocitest ..... ***Failed 0.
  ↵06 sec

```

```
      Start 56: pcltest
56/61 Test #56: pcltest ..... Passed 0.
↳ 28 sec
      Start 57: pgpointcloudtest
57/61 Test #57: pgpointcloudtest ..... Passed 1.
↳ 66 sec
      Start 58: plangtest
58/61 Test #58: plangtest ..... Passed 0.
↳ 14 sec
      Start 59: python_predicate_test
59/61 Test #59: python_predicate_test ..... Passed 0.
↳ 16 sec
      Start 60: python_programmable_test
60/61 Test #60: python_programmable_test ..... Passed 0.
↳ 15 sec
      Start 61: sqlitetest
61/61 Test #61: sqlitetest ..... Passed 0.
↳ 55 sec

97% tests passed, 2 tests failed out of 61

Total Test time (real) = 21.57 sec

The following tests FAILED:
  49 - pdal_plugins_cpd_kernel_test (OTHER_FAULT)
  55 - ocitest (Failed)
```

For a more verbose output, use the `-V` flag. Or, to run an individual test suite, use `-R <suite name>`. For example:

```
$ ctest -V -R pdal_io_bpf_test
```

Should produce output similar to:

```
UpdateCTestConfiguration from :/Users/hobu/dev/git/pdal/
↳ DartConfiguration.tcl
UpdateCTestConfiguration from :/Users/hobu/dev/git/pdal/
↳ DartConfiguration.tcl
Test project /Users/hobu/dev/git/pdal
Constructing a list of tests
```

```
Done constructing a list of tests
Checking test dependency graph...
Checking test dependency graph end
test 18
    Start 18: pdal_io_bpf_test

18: Test command: /Users/hobu/dev/git/pdal/bin/pdal_io_bpf_test
18: Environment variables:
18:   PDAL_DRIVER_PATH=/Users/hobu/dev/git/pdal/lib
18: Test timeout computed to be: 9.99988e+06
18: [=====] Running 20 tests from 1 test case.
18: [-----] Global test environment set-up.
18: [-----] 20 tests from BPFTest
18: [ RUN      ] BPFTest.test_point_major
18: [ OK       ] BPFTest.test_point_major (8 ms)
18: [ RUN      ] BPFTest.test_dim_major
18: [ OK       ] BPFTest.test_dim_major (3 ms)
18: [ RUN      ] BPFTest.test_byte_major
18: [ OK       ] BPFTest.test_byte_major (4 ms)
18: [ RUN      ] BPFTest.test_point_major_zlib
18: [ OK       ] BPFTest.test_point_major_zlib (6 ms)
18: [ RUN      ] BPFTest.test_dim_major_zlib
18: [ OK       ] BPFTest.test_dim_major_zlib (4 ms)
18: [ RUN      ] BPFTest.test_byte_major_zlib
18: [ OK       ] BPFTest.test_byte_major_zlib (5 ms)
18: [ RUN      ] BPFTest.roundtrip_byte
18: [ OK       ] BPFTest.roundtrip_byte (15 ms)
18: [ RUN      ] BPFTest.roundtrip_dimension
18: [ OK       ] BPFTest.roundtrip_dimension (10 ms)
18: [ RUN      ] BPFTest.roundtrip_point
18: [ OK       ] BPFTest.roundtrip_point (11 ms)
18: [ RUN      ] BPFTest.roundtrip_byte_compression
18: [ OK       ] BPFTest.roundtrip_byte_compression (16 ms)
18: [ RUN      ] BPFTest.roundtrip_dimension_compression
18: [ OK       ] BPFTest.roundtrip_dimension_compression (13 ms)
18: [ RUN      ] BPFTest.roundtrip_point_compression
18: [ OK       ] BPFTest.roundtrip_point_compression (14 ms)
18: [ RUN      ] BPFTest.roundtrip_scaling
18: [ OK       ] BPFTest.roundtrip_scaling (10 ms)
18: [ RUN      ] BPFTest.extra_bytes
```

```
18: [      OK ] BPFTest.extra_bytes (15 ms)
18: [ RUN    ] BPFTest.bundled
18: [      OK ] BPFTest.bundled (17 ms)
18: [ RUN    ] BPFTest.inspect
18: [      OK ] BPFTest.inspect (1 ms)
18: [ RUN    ] BPFTest.mueller
18: [      OK ] BPFTest.mueller (0 ms)
18: [ RUN    ] BPFTest.flex
18: [      OK ] BPFTest.flex (9 ms)
18: [ RUN    ] BPFTest.flex2
18: [      OK ] BPFTest.flex2 (7 ms)
18: [ RUN    ] BPFTest.outputdims
18: [      OK ] BPFTest.outputdims (14 ms)
18: [-----] 20 tests from BPFTest (182 ms total)
18:
18: [-----] Global test environment tear-down
18: [=====] 20 tests from 1 test case ran. (182 ms total)
18: [ PASSED ] 20 tests.
1/1 Test #18: pdal_io_bpf_test ..... Passed 0.20 sec
```

The following tests passed:
pdal_io_bpf_test

100% tests passed, 0 tests failed out of 1

```
$ bin/pdal_io_test
```

Again, the output should resemble the following:

```
[=====] Running 20 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 20 tests from BPFTest
[ RUN    ] BPFTest.test_point_major
[      OK ] BPFTest.test_point_major (7 ms)
[ RUN    ] BPFTest.test_dim_major
[      OK ] BPFTest.test_dim_major (3 ms)
[ RUN    ] BPFTest.test_byte_major
[      OK ] BPFTest.test_byte_major (4 ms)
```

```

[ RUN      ] BPFTest.test_point_major_zlib
[ OK      ] BPFTest.test_point_major_zlib (5 ms)
[ RUN      ] BPFTest.test_dim_major_zlib
[ OK      ] BPFTest.test_dim_major_zlib (5 ms)
[ RUN      ] BPFTest.test_byte_major_zlib
[ OK      ] BPFTest.test_byte_major_zlib (6 ms)
[ RUN      ] BPFTest.roundtrip_byte
[ OK      ] BPFTest.roundtrip_byte (17 ms)
[ RUN      ] BPFTest.roundtrip_dimension
[ OK      ] BPFTest.roundtrip_dimension (10 ms)
[ RUN      ] BPFTest.roundtrip_point
[ OK      ] BPFTest.roundtrip_point (11 ms)
[ RUN      ] BPFTest.roundtrip_byte_compression
[ OK      ] BPFTest.roundtrip_byte_compression (15 ms)
[ RUN      ] BPFTest.roundtrip_dimension_compression
[ OK      ] BPFTest.roundtrip_dimension_compression (14 ms)
[ RUN      ] BPFTest.roundtrip_point_compression
[ OK      ] BPFTest.roundtrip_point_compression (14 ms)
[ RUN      ] BPFTest.roundtrip_scaling
[ OK      ] BPFTest.roundtrip_scaling (11 ms)
[ RUN      ] BPFTest.extra_bytes
[ OK      ] BPFTest.extra_bytes (16 ms)
[ RUN      ] BPFTest.bundled
[ OK      ] BPFTest.bundled (17 ms)
[ RUN      ] BPFTest.inspect
[ OK      ] BPFTest.inspect (1 ms)
[ RUN      ] BPFTest.mueller
[ OK      ] BPFTest.mueller (0 ms)
[ RUN      ] BPFTest.flex
[ OK      ] BPFTest.flex (8 ms)
[ RUN      ] BPFTest.flex2
[ OK      ] BPFTest.flex2 (7 ms)
[ RUN      ] BPFTest.outputdims
[ OK      ] BPFTest.outputdims (14 ms)
[-----] 20 tests from BPFTest (185 ms total)

[-----] Global test environment tear-down
[=====] 20 tests from 1 test case ran. (185 ms total)
[ PASSED ] 20 tests.

```

This invocation allows us to alter Google Test's default behavior. For more on the available flags type:

```
$ bin/<test_name> --help
```

Key among these flags are the ability to list tests (`--gtest_list_tests`) and to run only select tests (`--gtest_filter`).

Test Data

Use the directory `./test/data` to store files used for unit tests. A function is provided in the `Support` class for referencing that directory in a configuration-independent manner.

Temporary output files from unit tests should go into the `./test/temp` directory. A `Support` function is provided for referencing this directory as well.

Unit tests should always clean up and remove any files that they create (except perhaps in case of a failed test, in which case leaving the output around might be helpful for debugging).

Continuous Integration

PDAL *regression tests* (page 411) are run on a per-commit basis by at least two continuous integration platforms.

Status

(<https://travis-ci.org/PDAL/PDAL>) (<https://ci.appveyor.com/project/hobu/pdal>)

Travis

The Travis continuous integration platform runs the PDAL test suite on Linux. The build status and other supporting information can be found at <https://travis-ci.org/PDAL/PDAL>. Its configuration can be found at <https://github.com/PDAL/PDAL/blob/master/.travis.yml>. All administrators of the GitHub *PDAL* group have rights to modify the Travis configuration.

It uses the `pdal/dependencies` *Install Docker* (page 9) image found at <https://hub.docker.com/r/pdal/dependencies> as a base platform for providing prerequisite

software and running the test suite. If you want to add new test functionality based on a dependency, you will need to update that Docker image to do so.

AppVeyor

PDAL uses the AppVeyor continuous integration platform to run the PDAL compilation and test suite on Windows. The build status and other supporting information can be found at <https://ci.appveyor.com/project/hobu/pdal>. Its configuration can be found at <https://github.com/PDAL/PDAL/blob/master/appveyor.yml>. All administrators of the GitHub *PDAL* group have rights to modify the AppVeyor configuration.

Howard Butler (<http://github.com/hobu>) currently pays the bill to run in the AppVeyor upper performance processing tier. The AppVeyor configuration depends on [OSGeo4W64](http://trac.osgeo.org/osgeo4w/) (<http://trac.osgeo.org/osgeo4w/>) for dependencies. If you want to add new test functionality based on a dependency, you will need to update OSGeo4W64 with a new package to do so.

API

PDAL is a C++ library, and its primary API is in that language. There is also a Python API (<https://pypi.python.org/pypi/PDAL>) that allows reading of data and interaction with [NumPy](http://www.numpy.org/) (<http://www.numpy.org/>).

C++ API

`pdal::BOX2D`

Warning: doxygenclass: Cannot find class “pdal::BOX2D” in doxygen xml output for project “api” from directory: doxygen/xml

Warning: doxygenclass: Cannot find class “pdal::BOX3D” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::Charbuf`

Warning: doxygenclass: Cannot find class “pdal::Charbuf” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::Dimension`

Warning: doxygennamespace: Cannot find namespace “pdal::Dimension” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::Extractor`

Warning: doxygenclass: Cannot find class “pdal::Extractor” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::FileUtils`

Warning: doxygennamespace: Cannot find namespace “pdal::FileUtils” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::Filter`

Warning: doxygenclass: Cannot find class “pdal::Filter” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::IStream`

Warning: doxygenclass: Cannot find class “pdal::IStream” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::Log`

Warning: doxygenclass: Cannot find class “pdal::Log” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::Metadata`

Warning: doxygenclass: Cannot find class “pdal::Metadata” in doxygen xml output for project “api” from directory: doxygen/xml

Warning: doxygenclass: Cannot find class “pdal::MetadataNode” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::Options`

Warning: doxygenclass: Cannot find class “pdal::Options” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::PointTable`

Warning: doxygenclass: Cannot find class “pdal::PointTable” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::PointView`

Warning: doxygenclass: Cannot find class “pdal::PointView” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::ProgramArgs`

Warning: doxygenclass: Cannot find class “pdal::ProgramArgs” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::Reader`

`pdal::Reader` 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`.

Warning: doxygenclass: Cannot find class “pdal::Reader” in doxygen xml output for project “api” from directory: doxygen/xml

`pdal::Stage`

`pdal::Stage` is the base class of `pdal::Filter`, `pdal::Reader`, and `pdal::MultiFilter` classes that implement the reading API in a PDAL pipeline.

Warning: doxygenclass: Cannot find class “pdal::Stage” in doxygen xml output for project “api” from directory: doxygen/xml

pdal::StageFactory

Warning: doxygenclass: Cannot find class “pdal::StageFactory” in doxygen xml output for project “api” from directory: doxygen/xml

pdal::Utils

:cpp:namespace:‘pdal::Utils’ is a set of utility functions.

Warning: doxygennamespace: Cannot find namespace “pdal::Utils” in doxygen xml output for project “api” from directory: doxygen/xml

pdal::Writer

Warning: doxygenclass: Cannot find class “pdal::Writer” in doxygen xml output for project “api” from directory: doxygen/xml

FAQ

- How do you pronounce PDAL?

The proper spelling of the project name is PDAL, in uppercase. It is pronounced to rhyme with “GDAL”.

- 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 *filterspclblock* (page 141) mechanism.

- 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 17) describes application operations you can achieve with PDAL.

- Is there any compatibility with libLAS's LAS Utility Applications or LAStools?

No. The the command line interface was developed from scratch with focus on usability and readability. You will find that the `pdal` command has several well-organized subcommands such as `info` or `translate` (see *Applications* (page 17)).

License

Unless otherwise indicated, all files in the PDAL distribution are

Copyright (c) 2015, 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.

Overall PDAL license (BSD)

Copyright (c) 2015, 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.

References

**CHAPTER
TEN**

INDICES AND TABLES

- genindex
- search

BIBLIOGRAPHY

- [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.
- [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, 278

B

boundary, 306

C

classification, 342

Clipping, 311

CloudCompare, 276

Colorization, 318

Command line, 278

coordinate system, 295

csd, 360

CSV, 294

D

Denoising, 323

Density, 328

density, 335

Docker, 9, 280

docker run, 293

DSM, 349

DTM, 349

E

elevation model, 349

Embed, 279

Entwine, 275

Extension, 279

F

Filtering, 323

filtering, 342

Fusion, 276

G

GDAL, 318

georeferencing, 269, 360

GeoWave, 52, 80

GNSS/IMU, 269, 360

Greyhound, 275

ground, 342

H

hexagon tessellation, 328

I

info command, 293

J

JSON, 294

L

LASTools, 274

libLAS, 276

M

metadata, 295

N

nearby, 296

nearest, 296

Numpy, 279

O

OGR, 306, 311, 328

Optech, 360

OrfeoToolbox, 276

P

PCL, 275

poisson, 335

pronounce, 425

Python, 279

Q

QGIS, 286, 306

query, 296

Quickstart, 9

R

Raster, 318

References, 427

Reprojection, 301

RGB, 318

Riegl, 360

S

sample, 335

search, 296

SOCS, 269

software installation, 280

spatial reference system, 295

Stage, 424

Start Here, 293

T

thinning, 335

U

Utils, 425

UTM, 301, 360

V

Vector, 311

voxel sampling, 335

W

WGS84, 301, 360