

PDAL for Point Cloud Processing

UAV OS HandOut 02

This handout will teach you how you can use powerful Point Data Abstraction Library (PDAL) for Point Cloud data processing and also helps in better understanding of the point clouds presented with regard to its statistics, metadata etc. The PDAL can be used to translate, segment, apply powerful filters, query or classify the point clouds using its attribute values. Finally, the reusable PDAL JSON based pipelines can be used to create DSM, DTMs etc



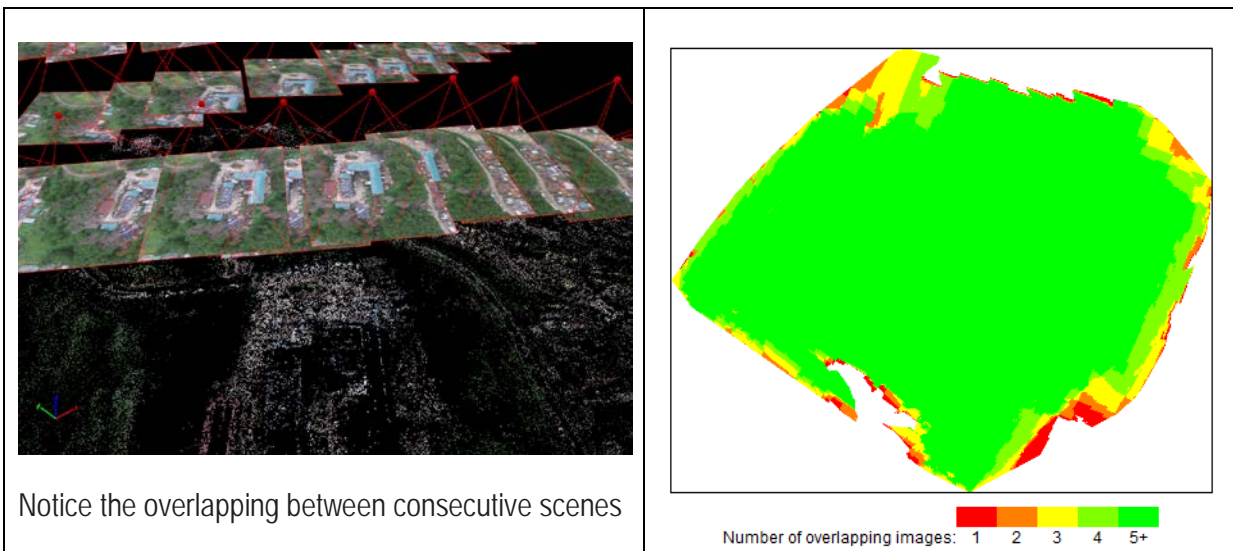
What You Will Learn

- Learn about the PDAL
- Examine and understand Point Cloud Data Structures
- Search and Query into Point Cloud Data
- Translate or re-project Point Clouds
- Use filters and algorithms to filter and segment or classify
- Build process workflow pipeline for generating product such as DSMs

Note: All commands has been highlighted in **red** for easy reference

About the Data: 3D Point Clouds

- **Sample1:** sample LiDAR survey data of Dublin (Ireland). You can download more such data from <https://geo.nyu.edu/catalog/nyu-2451-38625>
- **Sample2:** UAV Photogrammetry data of NESAC: Mixed scene (part of West Khasi Hills) covering a small area of about 9.6 ha. T600 DJI inspire series of UAV that has Zenmuse X3-FC350 camera with focal length of 4 mm and an effective resolution of 12.4 mega pixels giving a maximum image size of 4000x3000. This was flown at a height of 100m to collect about 42 images of the area.



Learning Goal 1: Understand PDAL and how it can be used for processing a given point cloud. Further, you'll learn basics of point clouds and their attributes or dimensions.

What is 'point cloud'?

- Set of $(x,y,z,r,i,...)$ measured points reflected from Earth surface or objects on or above it, where x,y,z may/may not be geo-referenced coordinates, r is the return number and i is intensity, $r:g:b$ may also be measured specially for photogrammetry based point clouds.
- Provided in generally

- ASCII (x,y,z,...) format
- Binary LAS format (header, record info, x,y,z,i, scan direction, edge of flight line, classification, etc
- PLY or Polygon File Format also called as Standard Triangle format. It can store variety of properties such as colours, surface normal, texture 3D coordinates etc.

Briefly, a point cloud is a collection of vertices in 3D space, described by X, Y and Z coordinates at each vertex. The Point cloud may or may not have a regular structure. It also may called as a cluster of three dimensional points and, often, associated attributes like colour information or the intensity of the return based on its source – digital camera or laser.

Why must we process Point Clouds

- Point clouds from different sources vary in their properties such as number of returns, density, or quality and therefore there are challenges in understanding and extracting useful information out of it.
- The point cloud in its original form is not quite useful. To make it really useful, we must process and transform it into ground models and representations of objects such as buildings and trees.
- We can process Point clouds for some of the task as follows
 - Filtering outliers (birds etc?)
 - Separating Ground or Non-Ground points (for DSM, DTMs)
 - Classification and extraction of Vegetation, Building Structures, Other rigid structures
 -



Figure: The above point cloud is derived via SfM approach and consists of points representing features such as vegetation, buildings, grounds, tall trees etc

How Point Clouds are generated

- Using methods such as Photogrammetric via SfM, it can generate point clouds which each point spatially encoded with X, Y and Z (with other attributes or dimensions) which are generally very large, unstructured datasets at with high density and spacing - at centimetre or so. Similarly LiDAR sensors can collect thousands of points per square volume.
- The structure of point cloud data generated by SfM being highly unstructured and thus don't exist on regular grids, it becomes challenging to process and analyse these.

Why LAS/LAZ format

This LAS format is a commonly used open data exchange standard by the American Society for Photogrammetry and Remote Sensing (ASPRS).

- LAS format is well known format for storing for point clouds derived both by LiDAR and Photogrammetry approach and good for analysis.
- This format is quite commonly used for performing information extraction analysis. Further, the utility such as LASzip compressor can be used for compressing the data and making data delivery and exchange more at ease.

What is PDAL?

- PDAL is Point Data Abstraction Library. It is a C/C++ open source library and applications for translating and processing point cloud data.
- PDAL allows you to compose operations on point clouds into pipelines of stages.
- Stages consist of readers, filters or writers.
- These pipelines can be written in a declarative JSON syntax or constructed using the available API.

Installing PDAL

Install Anaconda or Miniconda

><https://www.anaconda.com/distribution/#download-section>

Install PDAL from channel conda-forge on our environment named pdal

>conda create --yes --name pdal --channel conda-forge pdal

Update PDAL to the latest version(optional)

>conda update pdal

Activate conda

>conda activate pdal (for environment named pdal)

Deactivate conda

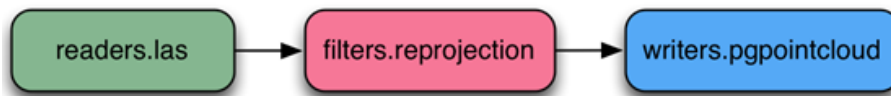
>conda deactivate

Why PDAL?

- PDAL is a set of applications and library to facilitate translation of point cloud data between various formats

- Provides some facilities for transformation of data between various geometric projections and can calculate some statistical, boundary and density data.
- PDAL also provides point classification algorithms such as PMF/SMRF
- PDAL allows for creation of reusable components which can be chained together as single process pipeline.
- PDAL provides an API that can be used by programmers for integration into their own projects or to allow extension of existing capabilities.

Understanding PDAL Process



- Pipelines consists of workflows which consists of composed of stages (readers/writers/filters) specified as JSON. JSON is a lightweight format for storing and transporting data.
- PDAL views point cloud processing operations as a pipeline composed as a series of stages. A stage is the PDAL way of describing a processing step which operates on points. Eg, Imagine a simple pipeline composed of a LAS Reader stage, a Reprojection stage, and a PostgreSQL Writer. Instead of writing a single and big one specialized program to perform this operation, we can dynamically compose it as a sequence of steps or operations.

A PDAL JSON Pipeline that composes this operation to re-project and load the data into PostgreSQL might look something like the following:

```

1 {
2   "pipeline":[
3     {
4       "type":"readers.las",
5       "filename":"input.las"
6     },
7     {
8       "type":"filters.reprojection",
9       "out_srs":"EPSG:4326"
10    },
11    {
12      "type":"writers.pgpointcloud",
13      "connection":"host=localhost dbname=nesac user=puyam",
14      "table":"output",
15      "srid":"4326"
16    }
17  ]
18}

```

- The Readers: Readers are a stage which reads in data (having atleast X , Y , and Z dimension) from a file system into a PDAL method called **pointview**. PDAL includes a range of 'built in' readers, and others can be added or removed as plugins at build time.
- The Filters: Once you've read data into a pointview, filters (methods) are another stage where the real point data processing happens! We can also stack filters together to form complex one line commands - with the pointview modified by

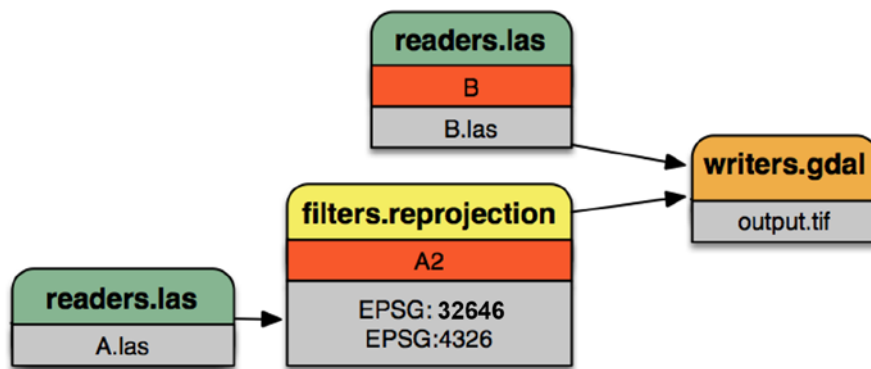
one filter being passed to the next or tagged to be reference by other stages. Each filter works by modifying the view, in order to get the desired output.

- The writers: Writers are a stage which writes out a pointview to a file, database or whatever!
- **In Short** : **Input** Readers (File/Database) -> **Intermediate** Filters/Algorithms (PointView) -> **Output** Writers (File/Database)

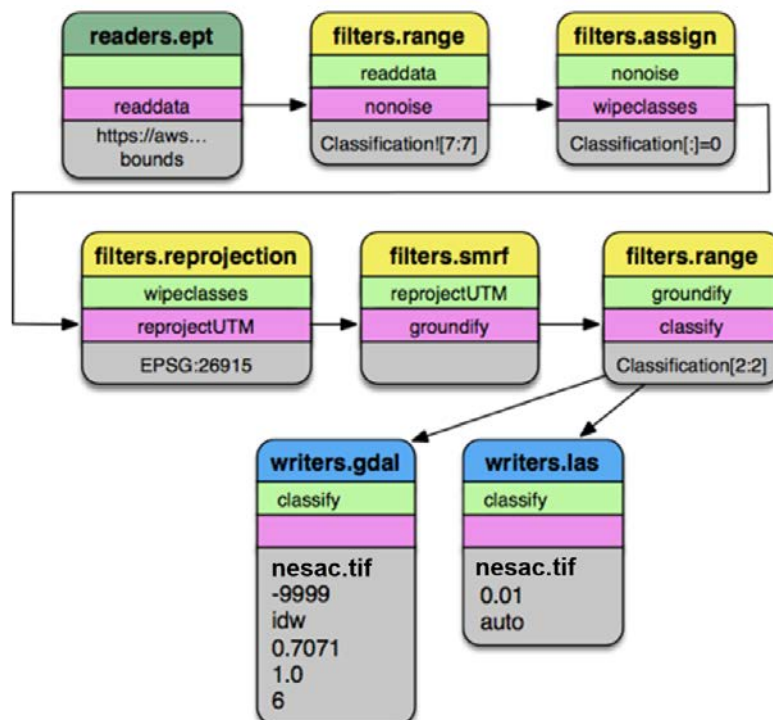
For example:

`pdal translate -i input_data.las -o output_data.las`

The above PDAL one liner expresses a reader, filter, and writer to translate point data from one format to another.



Simple PDAL Pipeline depicting re-projection

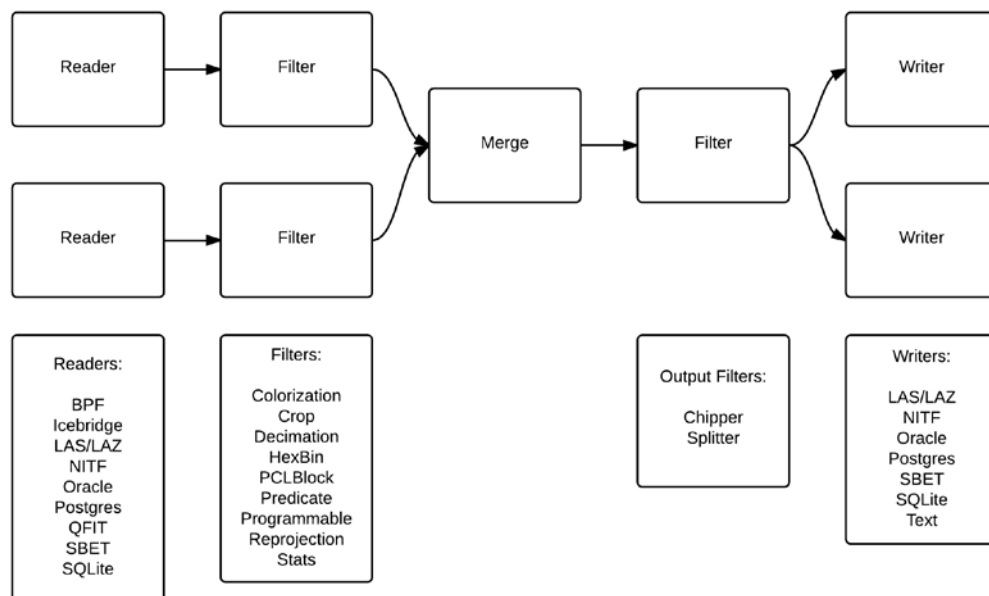


Advance PDAL Pipeline depicting filtering and classification of Point Clouds

- 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, and it provides a command, pipeline, to allow you to execute them.
- PDAL is an open source project, with all of its development activities available online at <https://github.com/PDAL/PDAL>

A Brief Architecture of PDAL

An overview of PDAL Architecture is given below – PDAL reads data from a set of input sources using format-specific readers. Point data can be passed through various filters that transform data or create metadata. If desired, points can be written to an output stream using a format-specific writer. PDAL can merge data from various input sources into a single output source, preserving attribute data where supported by the input and output formats.



The above diagram shows a possible arrangement of PDAL readers, filters and writers, all of which are known as stages. Any merge operation or filter may be placed after any reader. Output filters are distinct from other filters only in that they may create more than one set of points to be further filtered or written. The arrangement of readers, filters and writers is called a PDAL pipeline. Pipelines can be specified using JSON.

A point view (PointView object) stores references to points. Storage and retrieval of points is done through a pointView 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.

In PDAL, dimensions are the set of things described in the point cloud data schema. For LAS format files, dimensions are set within ASPRS LAS standard bounds. Read more on LAS format here (https://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf). Some common dimensions are:

Name	Description
X	'X' coordinate (longitude in EPSG:4326)
Y	'Y' coordinate (latitude in EPSG:4326)

Z	'up' coordinate
Intensity	Laser return intensity
Red	Red channel in RGB colour
Green	Green channel in RGB colour
Blue	Blue channel in RGB colour

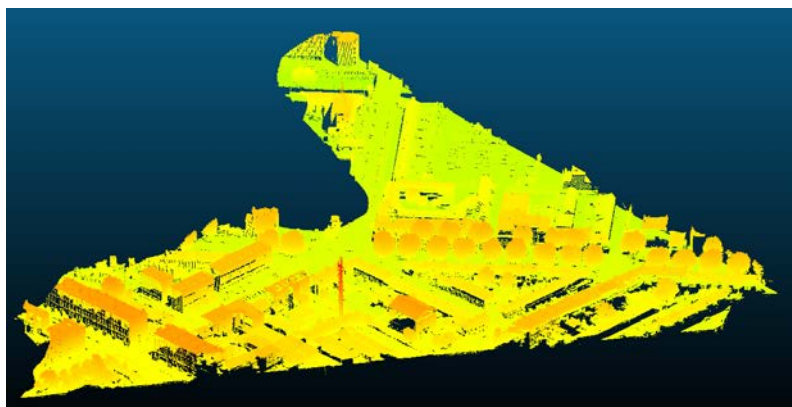
The list of standard PDAL dimensions is here: <https://pdal.io/dimensions.html> - again, many of these are derived from ASPRS LAS formats. Primarily because it's a data exchange format we can't ignore, and second - it's a standard, and while never perfect, standards make implementation easier. Note: PDAL thinks in stages and point views. Readers to create views, filters to operate on views, and writers to write views out.

Now, with these knowledge, let's try processing of the given point clouds

[A] Using PDAL on free sample lidar data-1

Learning Goal 2: Use Command line processing of PDAL to query into the data and know about the point data in terms of what's in it, where it is, and whether there are any problems with it

Below shows a glimpse of data visualized in **CloudCompare** which also happens to be is a 3D point cloud processing software:



pdal info DUBLIN.laz

The above simple command outputs (a part of the output shown on the right) a huge JSON attributes, including summary statistics for each dimension in the file. Further, it also shows the projection information in EPSG and its extent.

Output (A part of)

```
{
  "bbox": {
    "EPSG:4326": {
      "bbox": {
        "maxx": -6.250113388,
        "maxy": 53.36043314,
        "maxz": 59.915,
        "minx": -6.257095951,
        "miny": 53.35687648,
        "minz": -71.555
      },
      "boundary": {
        "coordinates": [
          [
            -6.2570959510635058,
            53.356976601776466,
            -71.555000000000007
```


Let's look at some different metadata queries. Observe each of the output and note the differences:

```
pdal info DUBLIN.laz --stats
pdal info DUBLIN.laz --metadata
pdal info DUBLIN.laz --summary
pdal info DUBLIN.laz --boundary
```

Searching for points

We can also look for information about specific points.

Try: `pdal info DUBLIN.laz -p 0`

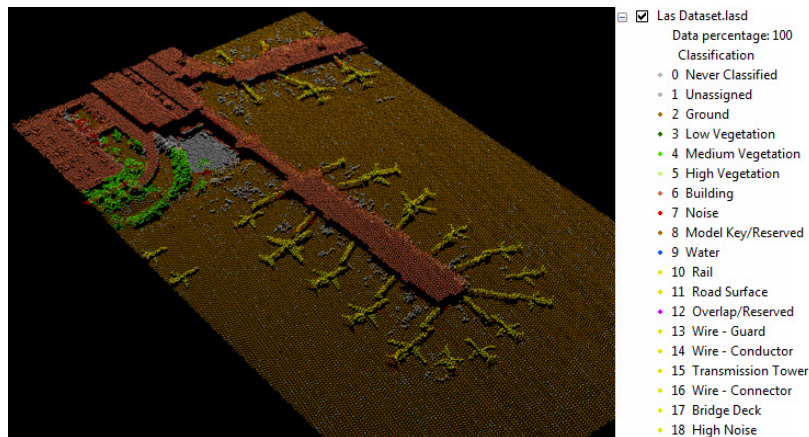
The output should tell you all about the first point (0-indexed) in the file. Every attribute value here means several things. Please refer to the description table below

```
{
  "filename": "DUBLIN.laz",
  "pdal_version": "2.0.1 (git-
version: Release)",
  "points":
  {
    "point":
    {
      "Classification": 2,
      "EdgeOfFlightLine": 0,
      "GpsTime": 394488.1361,
      "Intensity": 11,
      "NumberOfReturns": 3,
      "PointId": 0,
      "PointSourceId": 21,
      "ReturnNumber": 3,
      "ScanAngleRank": 32,
      "ScanDirectionFlag": 1,
      "UserData": 0,
      "X": 316266.358,
      "Y": 235846.347,
      "Z": -62.947
    }
  }
}
```

Before start doing anything, let's first understand the point cloud attributes for LAS/LAZ format. The following table describes about the attributes of each point (for las file attributes):

Attribute Name	Description
Intensity	The return strength will tell more about the physical characteristics of the object . It tells the on the reflectivity of the object on or above the ground. This value can also be used for feature detection and extraction which can help in classification of the point cloud.
Return number	An emitted laser pulse can have up to five returns depending on the features it is reflected from and the capabilities of the laser scanner used to collect the data. The first return will be flagged as return number one, the second as return number two, and so on.

Attribute Name	Description
Number of returns	The number of returns is the total number of returns for a given pulse . For example, a laser data point may be return two (return number) within a total number of five returns.
Point classification	<p>Every lidar point that is post-processed can have a classification that defines the type of object that has reflected the laser pulse. Lidar points can be classified into a number of categories including bare earth or ground, top of canopy, and water etc. The different classes are defined using numeric integer codes in the LAS files. Classification codes were defined by ASPRS (American Society for Photogrammetry and Remote Sensing) for LAS formats 1.1, 1.2, 1.3, and 1.4.</p> <p>See below image for more on classification.</p>
Edge of flight line	The points will be symbolized based on a value of 0 or 1. Defines the points that falls at the edge of the flight line will be given a value of 1, and all other points will be given a value of 0.
RGB	Lidar data can be attributed with RGB (red, green, and blue) bands. This attribution often comes from imagery collected at the same time as the lidar survey.
GPS time	The GPS time stamp at which the laser point was emitted from the aircraft. The time is in GPS seconds of the week. Calculated/Counts started after defining Week Number and Day of the Week.
Scan angle	The scan angle is a value in degrees between -90 and +90. At 0 degrees, the laser pulse is directly below the aircraft at nadir. At -90 degrees, the laser pulse is to the left side of the aircraft, while at +90, the laser pulse is to the right side of the aircraft in the direction of flight. Most lidar systems are currently less than ±30 degrees .
Scan direction	The scan direction is the direction the laser scanning mirror was traveling at the time of the output laser pulse. A value of 1 is a positive scan direction, and a value of 0 is a negative scan direction. A positive value indicates the scanner is moving from the left side to the right side of the in-track flight direction, and a negative value is the opposite



Now, having understood the various attributes of LAS/LAZ point attributes, let's continue with our searching query.

Querying for points

Query for sets of points near ground control to assess accuracy. Here the **info** and **query** can be clubbed together to define a geographic extent. The command below will return the 3 nearest points to the coordinates (or known GCP?) provided (/3)

```
pdal info DUBLIN.laz --query "316200, 235400, 21/3"
```

Similarly, `pdal info DUBLIN.laz --query "316200, 235400, 21/8"` will return coordinates and dimension information for the 8 nearest points as a JSON attributes.

Learning Goal 3: Learn simple transformation tool for translating the input file by re-projection or compression etc.

PDAL can do straightforward data translation as a one liner using the translate convenience application. We'll inspect some use cases here. The basic pattern is : `pdal translate infile outfile --operation(s)`

Example: To compress a file (LAS) to LAZ format, try:

```
pdal translate -i infile.las -o outfile.laz (or reverse to uncompress) - which can also be fully expressed as: pdal translate -i infile.las -o outfile.laz --writers.las.compression=true
```

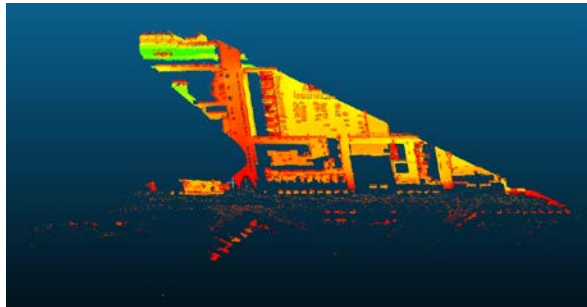
Re-projecting data is also a useful tool. This moves data from EPSG:29902 to EPSG:32629

```
pdal translate filters.reprojection -i DUBLIN.laz -o outfile.laz --filters.reprojection.in_srs=EPSG:29902 - --filters.reprojection.out_srs=EPSG:32629
```

Learning Goal 4: Learn the powerful filtering and cropping which can be as an intermediate tool. The filters are the most important stage in the entire process pipeline. It can perform clustering, assign value, removal of outliers or noise, classification of point clouds etc.

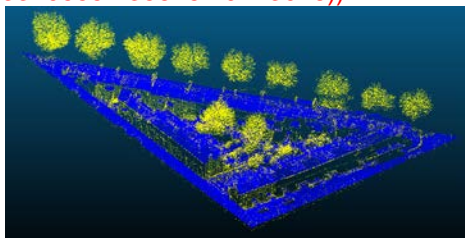
Points can be filtered many ways. We can restrict ranges point dimensions. If points carry labels (for example ASPRS classifications) we can use PDAL to exclude classes. We can also try to label 'noise' or points. The below command limits elevations between 0 and 10m above the reference surface. (More on filters will be given on subsequent page):

```
pdal translate filters.range -i DUBLIN.laz -o outfile.laz --filters.range.limits="Z[0:10]"
```



We can also subset points by geometry using a WKT polygon, shown here. Using the WKT polygon, we can crop and retrieve subset of points (coloured by classification*):

```
pdal translate filters.crop -i DUBLIN.laz -o outfile.laz --filters.crop.polygon="POLYGON
((316261.303310555 235626.19273016,316328.453964166 235522.105517281,316399.983804495
235578.043293381,316261.303310555 235626.19273016))"
```



* Scalar field > Active (Classification) (in CloudCompare)

PDAL can run as a command line application - using options to specify and modify stages and point views. This is really powerful, but can get too complex for long operations. That's why, we'll use pipelines to chain together these commands.

Learning Goal 5: Learning to create process workflow using Pipelines

What we've typed on the terminal is really invoking a PDAL pipeline in the background - which users a **reader** to ingest data to a **pointview** which can be operated on by filters, then writes out using a **writer**. We can also cast these workflows into JSON configuration snippets, and run them using the pipeline application.

Using reprojection as an example, let's create the JSON code snippet as follows:

```
{
  "pipeline":[
    {
      "type":"readers.las",
      "filename":"farm-sample.laz"
    },
    {
      "type":"filters.reprojection",
      "in_srs":"EPSG:32756",
```

The command line equivalent is:

```
pdal translate farm-sample.laz
farm-sample-reprojected.laz --
filters.reprojection.in_srs="EPSG:3
2756" --
filters.reprojection.out_srs="EPSG:
28356"
```

However - instead of using increasingly long command line processes,

<pre> "out_srs": "EPSG:28356" }, { "type": "writers.las", "compression": "laszip", "filename": "farm-sample- reprojected.laz" }] } } </pre> <p>Save the file as <i>reprojection.json</i> and run:</p> <p>pdal pipeline reprojection.json</p>	<p>the pipeline application allows creation of a library of standard processing tasks as easily-readable JSON files. Further, the already saved code snippet can be made as re-usable.</p>
--	--

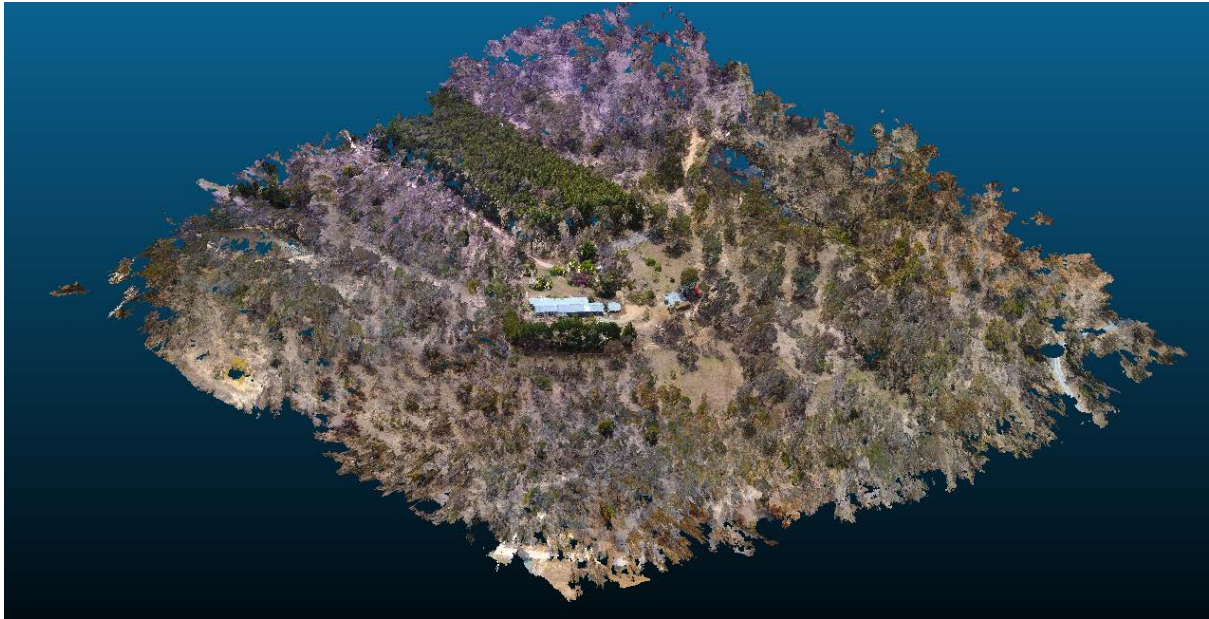
Learning Goal 6: Now, let's apply the knowledge learnt to actually classify a given point cloud.

Classifying ground points is a fundamental task for point cloud processing. Classification is hard, specially if the point cloud contains a mixture of terrain types and structure objects. Further, many photogrammetric point clouds won't have ground labels attached to points.

But, we can exploit a filter called SMRF (simple morphological filter) algorithm to try and classify the ground points and non-ground points. The algorithm consists of four conceptually distinct stages. The algorithms work in 4 steps:

- The first is the creation of the minimum surface (Zlmin).
- The second is the processing of the minimum surface, in which grid cells from the raster are identified as either containing bare-earth (BE) or objects (OBJ). This second stage represents the heart of the algorithm.
- The third step is the creation of a DEM from these gridded points.
- The fourth step is the identification of the original LIDAR points as either BE or OBJ based on their relationship to the interpolated DEM

We'll now try ground labelling for UAV derived data using the **sample farm-sample.laz**. Here's what the data look like:



Step1 : List out available Classification types with point cloud, if available:
`pdal info --stats farm-sample.laz --filters.stats.dimensions=Classification --filters.stats.enumerate=Classification`

As noticed, this point cloud has no classification labels! Let's try to fix that. Create a file 'rpas-ground.json' and populate it with:

```
{
  "pipeline":[
    {
      "type":"readers.las",
      "filename":"farm-sample.laz"
    },
    {
      "type":"filters.assign",
      "assignment":"Classification[:] = 0"
    },
    {
      "type":"filters.elm"
    },
    {
      "type":"filters.assign", "assignment":"NumberOfReturns[:] = 1"
    },
    {
      "type":"filters.assign", "assignment":"ReturnNumber[:] = 1"
    },
    {
      "type":"filters.outlier"
    },
    {
      "type":"filters.smrf"
    },
    {
      "type":"filters.range",
      "limits":"Classification[2:2]"
    },
    {
      "type":"writers.las",
```

```

    "filename": "farm-ground-default.laz"
  }
]
}

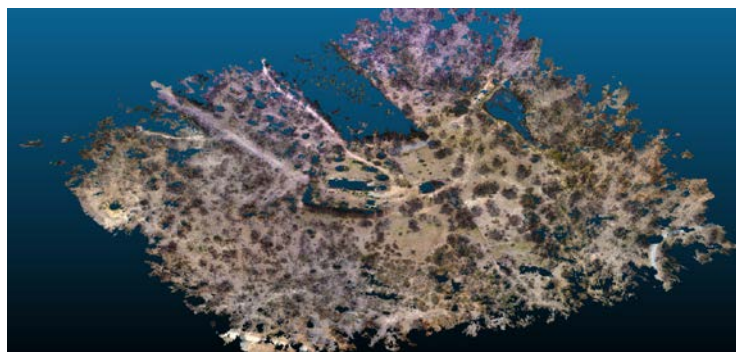
```

Save it, then run: `pdal pipeline rpas-ground.json`

As you can see, we've now stacked up a lot of filters together. Now, you can clearly see the convenience of using the pipeline approach than the usual command line approach. Lets now understand each filters used:

- `filters.assign` is used to label all points as **unclassified**
- `filters.elm` (extended local minimum) to label 'low points' as noise (ASPRS LAS class 7)
- `filters.outlier`, a statistical approach to label remaining outlier points as noise (ASPRS LAS class 7)
- `filters.smrf` (Simple Morphological Filter) to label points as 'ground', ignoring any points already labelled as 'noise'
- Finally, removing or limiting any points not labelled as ground from the output and writing them out to farm-ground-default.laz

Once you've got an output file, use CloudCompare (or another LAS/LAZ viewer), open farm-ground-default.laz and check the results:



Notice here that, only points labelled as 'ground' are returned - we've dropped any noise and unclassified points using a range filter. It's also **not the best segmentation** of 'ground' points - a stand of trees has been mislabelled! It needs further tweaking with the parameters defined in SMRF algorithms (see below). But, then we now know the powerful pipelines with its long lines of chaining with numerous filters to perform a task.

(Optional) Try modifying the pipeline and running parts of it, or removing some of the filters and see what happens by viewing results in CloudCompare.

Learning Goal 7: We will further dig down to filters, specially on SMRF algorithm filters for better classifying the point cloud and make better ground points.

Let's modify our pipeline a little to remove the final filter, and write out the entire dataset with noise and ground points labelled. We will then pass in some non-default filters.smrf options. Write the next JSON block out as **rpas-ground-allthepoints.json**

```

{
  "pipeline": [

```

```

{
  "type": "readers.las",
  "filename": "farm-sample.laz"
},
{
  "type": "filters.assign",
  "assignment": "Classification[:] = 0"
},
{
  "type": "filters.elm"
},
{
  "type": "filters.assign", "assignment": "NumberOfReturns[:] = 1"
},
{
  "type": "filters.assign", "assignment": "ReturnNumber[:] = 1"
},
{
  "type": "filters.outlier"
},
{
  "type": "filters.smrf",
  "ignore": "Classification[7:7]"
},
{
  "type": "writers.las",
  "filename": "ground-smrf-allthepoints.laz"
}
]
}

```

The SMRF algorithms provides sets of parameters for better classifying ground points. The parameters have been generated after considering wide variety of UAV data scenes. These parameters define a baseline set of general parameters which could be expected to give a reasonable performance for any given sample, and as such could be used as the starting point to tune the performance of the SMRF algorithm. **In order to measure the best-case performance of the simple progressive morphological filter, we can systematically vary 4 main inputs (slope, maximum window radius, and elevation threshold and scaling factor for ground identification).**

The SMRF algorithm applies a series of operations against a Point clouds representing digital surface model thereby creating a gridded model of the ground surface and a vector of boolean values for each tuple (x,y,z) describing it as either ground (0) or object (1).

- **SMRF must called with a slope threshold value (s), and a maximum window size (w).**
 - The **slope threshold** roughly corresponds to the **maximum slope of the terrain** in the given point cloud.
 - The **maximum window size** defines a window radius (in map units), and corresponds to the of **size largest feature to be removed**.

Sample	Optimized								Single parameter	
	Slopetol. (dz/dx)	Window radius (m)	Elevation threshold (m)	Scaling factor	T.I (%)	T.I.I (%)	T.E (%)	K (%)	T.E (%)	K (%)
1 (1-1)	0.20	16	0.45	1.20	7.88	8.81	8.28	83.12	8.64	82.40
2 (1-2)	0.18	12	0.30	0.95	2.57	3.30	2.92	94.15	3.10	93.80
3 (2-1)	0.12	20	0.60	0.00	0.26	4.07	1.10	96.77	1.88	94.43
4 (2-2)	0.16	18	0.35	1.30	2.57	5.07	3.35	92.21	3.40	92.07
5 (2-3)	0.27	13	0.50	0.90	3.21	6.17	4.61	90.73	6.48	87.02
6 (2-4)	0.16	8	0.20	2.05	2.25	6.90	3.52	91.13	4.19	89.49
7 (3-1)	0.08	15	0.25	1.50	0.39	1.52	0.91	98.17	2.48	95.00
8 (4-1)	0.22	16	1.10	0.00	3.64	8.17	5.91	88.18	10.79	78.41
9 (4-2)	0.06	49	1.05	0.00	0.27	1.98	1.48	96.48	2.93	93.07
10 (5-1)	0.05	17	0.35	0.90	0.59	4.44	1.43	95.76	3.00	90.74
11 (5-2)	0.13	13	0.25	2.20	3.09	10.08	3.82	81.04	4.17	78.80
12 (5-3)	0.45	3	0.10	3.80	1.18	31.97	2.43	68.12	7.41	47.24
13 (5-4)	0.05	11	0.15	2.30	2.51	2.05	2.27	95.44	3.67	92.65
14 (6-1)	0.28	5	0.50	1.45	0.51	10.70	0.86	87.22	2.02	75.38
15 (7-1)	0.13	15	0.75	0.00	0.99	6.84	1.65	91.81	1.85	90.52
Mean					2.13	4.47	2.97	90.02	4.40	85.40
Median					2.25	6.17	2.43	91.81	3.40	90.52
Min					0.26	1.52	0.86	68.12	1.85	47.24
Max					7.88	31.97	8.28	98.17	10.79	95.00
Std					1.99	7.37	2.07	7.85	2.70	12.34

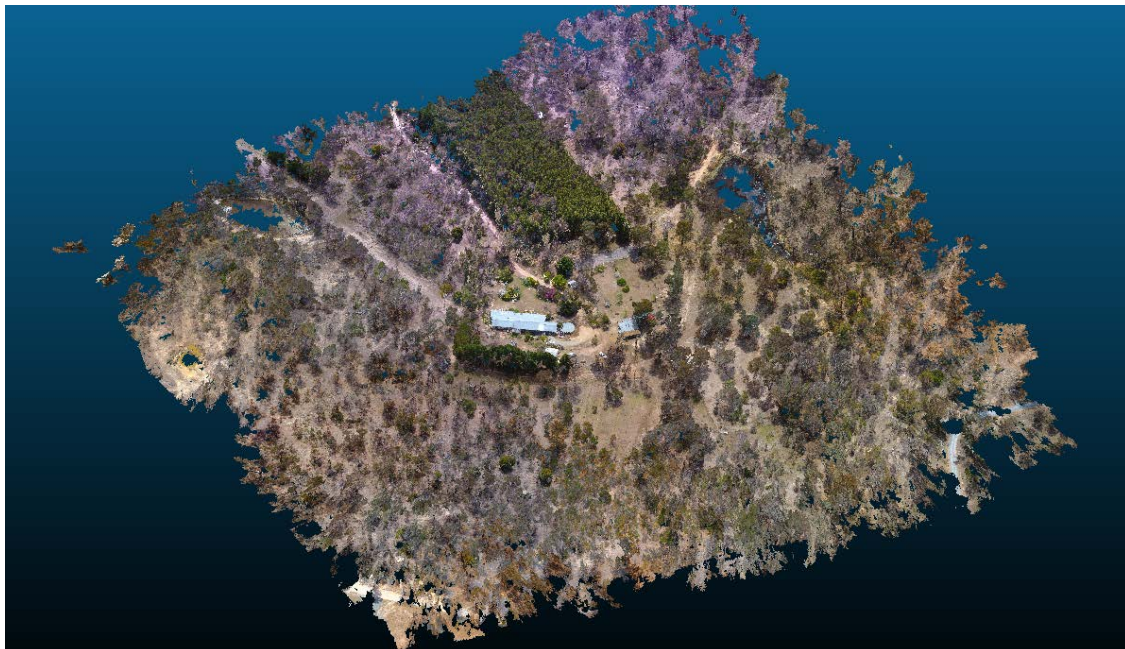
Optimized and single parameter results of the SMRF algorithm when tested against the ISPRS reference dataset expressed in Type I error rate (T.I), Type II error rate (T.II), total error rate (T.E), and Kappa (K, %). Single parameter results were obtained by using values (.15, 18, 0.5, 1.25) for slope tolerance, window radius, elevation threshold and scaling factor, respectively. All results used a one meter cell size to generate the Digital Surface and Elevation Models.

Now, invoke PDAL with some custom options to filters.smrf.

The set used here were obtained by trial-and-error - experiment and see what changes you get:

```
pdal pipeline rpas-ground-allthepoints.json --filters.smrf.slope=0.1 --filters.smrf.window=30 --filters.smrf.threshold=0.4
```

Visualising the ground points from this process, we see that we've managed to remove the unwanted treetops:

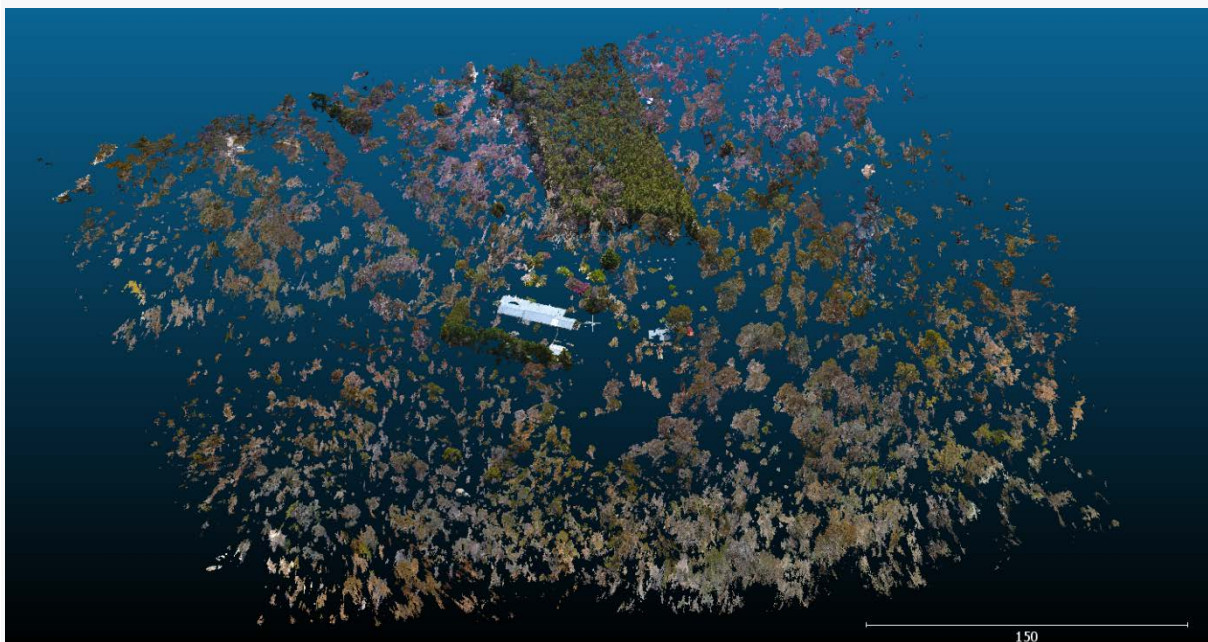


To happen this, we told filters.smrf that our ground is flatter than the default option (slope=0.1), our non-ground items might be wider than the default expectation (window=30), and our noise level might be a little higher (threshold=0.4). In short, any option from the stages used in the pipeline can be over-ridden by passing equivalent command line options.

Note : When framing pipelines, try to optimise them such that options which *need* to change often are as few as possible.

Also, try the following pipeline that *extracts all the points for above the ground*:

```
{
  "pipeline": [
    "farm-sample.laz",
    {
      "type": "filters.assign",
      "assignment": "Classification[:] = 0"
    },
    {
      "type": "filters.assign",
      "assignment": "NumberOfReturns[:] = 1"
    },
    {
      "type": "filters.assign",
      "assignment": "ReturnNumber[:] = 1"
    },
    {
      "type": "filters.smrf"
    },
    {
      "type": "filters.hag"
    },
    {
      "type": "filters.range",
      "limits": "HeightAboveGround[2:]"
    },
    {
      "filename": "above-ground.laz"
    }
  ]
}
```

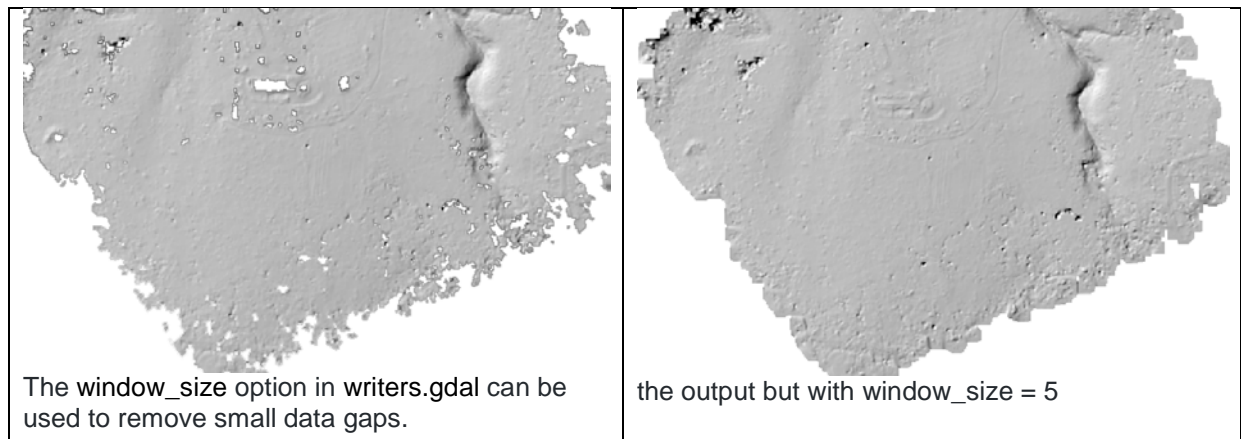


Learning Goal 8: Finally, we'll use PDAL process pipeline to actually generate a product, DTMs in this example.

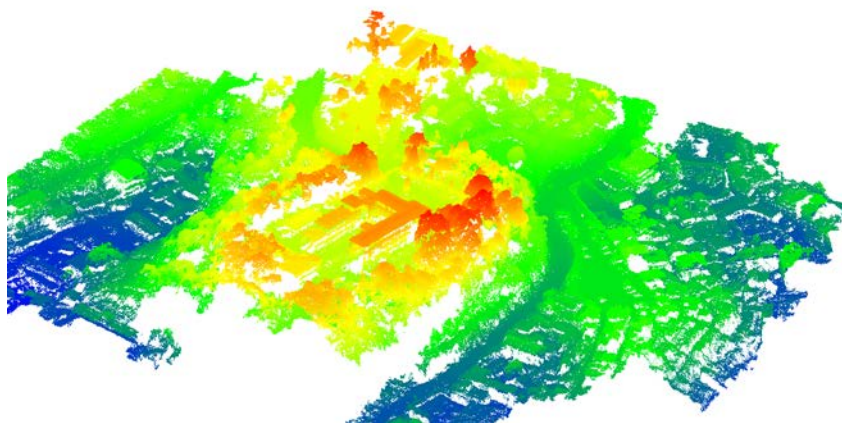
We'll now create a DTM from the ground points we just created. Also, since we've experimentally determined good values for ground segmentation, we'll put those in the pipeline JSON:

```
{
  "pipeline":[
    {
      "type":"readers.las",
      "filename":"farm-sample.laz"
    },
    {
      "type":"filters.assign",
      "assignment":"Classification[:] = 0"
    },
    {
      "type":"filters.elm"
    },
    {
      "type":"filters.assign", "assignment":"NumberOfReturns[:] = 1"
    },
    {
      "type":"filters.assign", "assignment":"ReturnNumber[:] = 1"
    },
    {
      "type":"filters.outlier"
    },
    {
      "type":"filters.smrf",
      "ignore":"Classification[7:7]",
      "slope":0.1,
      "window":60,
      "threshold":0.4
    },
    {
      "type":"filters.range",
      "limits":"Classification[2:2]"
    },
    {
      "type":"writers.gdal",
      "filename":"farm-dtm.tiff",
      "resolution":1,
      "output_type":"idw"
    }
  ]
}
```

...you can open the result in QGIS and take a look. Here's a preview:



Learning Goal 9: We'll now repeat and experiment the above commands on our photogrammetry derived Point cloud - sample data2



The above LAS file has been created using SfM and MVS techniques using 42 images from an area located in West Khasi Hills district, Meghalaya. Try running similar commands and state the difference you observed with the LiDAR point cloud we used in the above exercise.

```
pdal info wkhsourc2019.laz
pdal info wkhsourc2019.laz --stats
pdal info wkhsourc2019.laz --metadata
pdal info wkhsourc2019.laz --summary
pdal info wkhsourc2019.laz --boundary
pdal info wkhsourc2019.laz -p 0
```

```
{
  "filename": "wkhsourc2019.laz",
  "pdal_version": "2.0.1 (git-version: Release)",
  "points":
  {
    "point":
    {
      "Blue": 23296,
      "Classification": 0,
      "EdgeOfFlightLine": 0,
      "GpsTime": 0,
```

```

"Green": 26880,
"Intensity": 0,
"NumberOfReturns": 0,
"PointId": 0,
"PointSourceId": 0,
"Red": 30720,
"ReturnNumber": 0,
"ScanAngleRank": 0,
"ScanDirectionFlag": 0,
"UserData": 0,
"X": 325334.045,
"Y": 2823931.609,
"Z": 1377.158
}
}
}

```

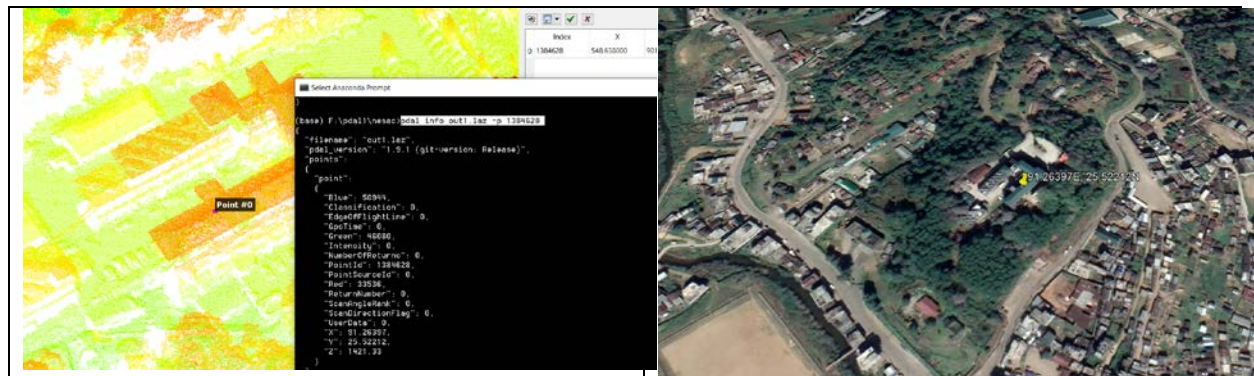
Notice now, on the point cloud being generated based on SfM photogrammetry approach, all all point clouds attributes are 0 except the X,Y,Z and R,G,B!

- Now, try further with finding points by geographic location. The query option to info can help: `pdal info wkhsourse2019.laz --query "325348, 2823943, 1380/3"`
As before, this will return the three nearest points to the coordinates provided (/3).
- Now, let's try re-projecting data which moves data from UTM Zone 46 to GCS Lat/Long as follows:

<p>Coordinate info for Point 0 in EPSG:32646 (before reprojection): <code>pdal info wkhsourse2019.laz -p 0</code> Output: "X": 325334.045,"Y": 2823931.609, "Z": 1377.158</p>	<p>Coordinate info for Point 0 in EPSG:4326 (after reprojection): <code>>pdal translate filters.reprojection -i wkhsourse2019.laz -o outfile.laz -- filters.reprojection.in_srs=EPSG:32646 -- filters.reprojection.out_srs=EPSG:4326</code> <code>>pdal info outfile.laz -p 0</code> Output: "X": 91.26, "Y": 25.52,"Z": 1377.158</p> <p>The current translate commands gives the X, Y values limited upto 2 decimal places. To extend it upto 5 decimal places, modify the above command as follows:</p> <pre> >pdal translate filters.reprojection -i wkhsourse2019.laz -o outfile.laz -- filters.reprojection.in_srs=EPSG:32646 -- filters.reprojection.out_srs=EPSG:4326 -- writers.las.scale_x=0.00001 -- writers.las.scale_y=0.00001 >pdal info outfile.laz -p 0 </pre> <p>Output: "X": 91.26394,"Y": 25.52213</p>
--	--

Bonus Exercise! : Get the specific point information, Get the Lat/Long details and examine it on Google Earth! And verify its positioning!
First, you may load the given point cloud to Cloud compare. Second, identify a point ID, say its

1384628. Third, use this point ID to get its geographic location information, say X,Y,Z. Once, noted, open Google Earth and check its position! Simple as that! ;-)



Filtering points

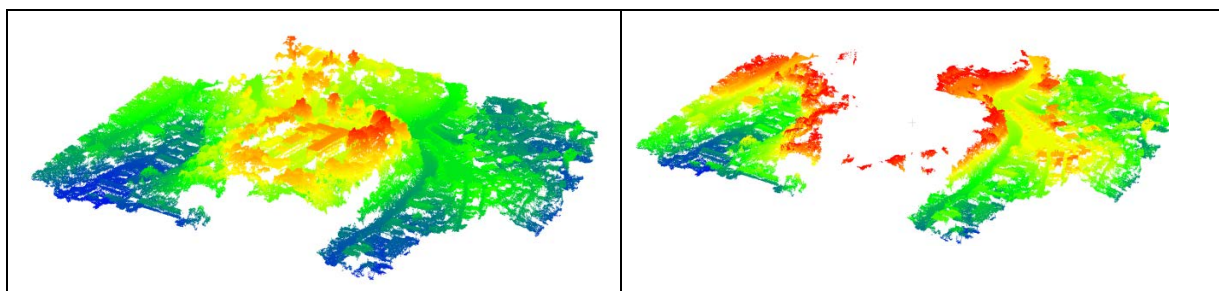
We can now continue running filter operations on our point cloud. Here in this example, we wish to limit elevations between 1369m and 1400m above the reference surface, (above sea level).

Before issuing the filtering operations, get the overall range of values for each attributes with stats command : `pdal info outfile.laz --stats`

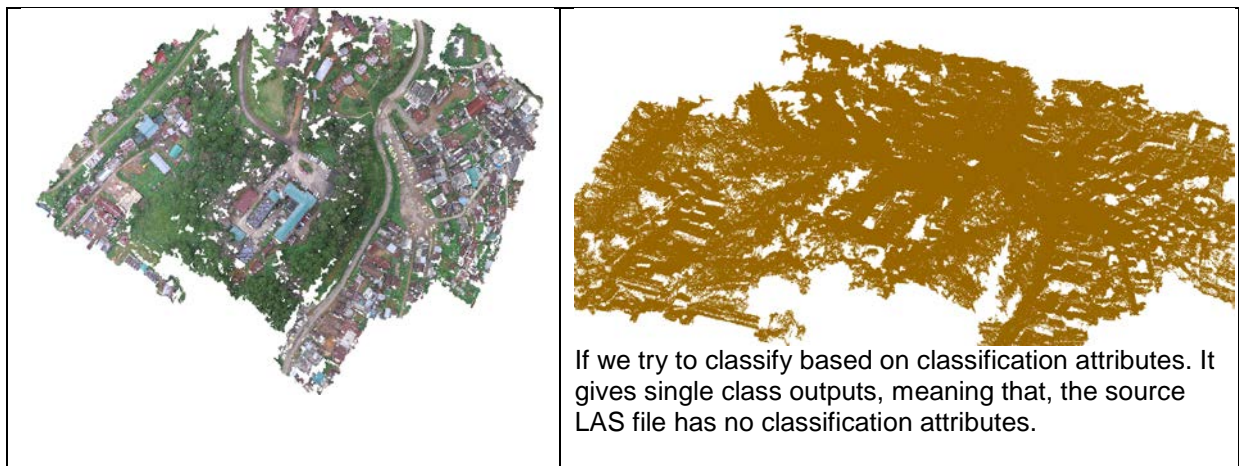
For elevations (Z), the following results shows the detail attributes for parameter Z : {"average": 1398.607912,"count": 3265688,"maximum": 1432.71, "minimum": 1369.56,"name": "Z", "position": 2, "stddev": 13.93626079, "variance": 194.2193648 }

`pdal translate filters.range -i wkhsourc2019.laz -o out2elev.laz filters.range.limits="Z[1369:1399]"`

Note : Use original source LAZ file (EPSG:32646)



We'll now do ground labelling for Point Cloud derived via SfM approach. The data looks like below:



Now, let's again create a PDAL JSON process pipeline to classify this. You may also use previously created json file to modify and run this. Below states 2 such methods for classifying. For detail, pl refer the explanation as given for sample data 1.

Method 1	Method 2
<pre> { "pipeline":[{ "type":"readers.las", "filename":"wkhsourc2019.laz" }, { "type":"filters.assign", "assignment":"Classification[:] = 0" }, { "type":"filters.elm" }, { "type":"filters.assign", "assignment": "Number of Returns[:] = 1" }, { "type":"filters.assign", "assignment": "Return Number[:] = 1" }, { "type":"filters.outlier" }, { "type":"filters.smrf" }, { "type":"filters.range", "limits":"Classification[2:2]" },] } </pre>	<pre> { "pipeline":["wkhsourc2019.las", { "type": "filters.outlier", "mean_k": 8, "multiplier": 3 }, { "type":"filters.assign", "assignment": "Number of Returns[:] = 1" }, { "type":"filters.assign", "assignment": "Return Number[:] = 1" }, { "type":"filters.smrf", "ignore": "Classification[7:7]" }, { "type":"filters.range", "limits": "Classification[2:2]" }, { "type":"writers.las", "filename":"wkh-ground.las" }] } </pre>


```

    "type": "writers.las",
    "filename": "wkhsourse2019-ground-
default.laz"
  }
]
}

```



As observed from the output above, only points labelled as 'ground' are returned - by dropping any noise and points which are unclassified using a range filter.

Generating DTM from Point Cloud

The following pipeline should create a DTM raster.

```

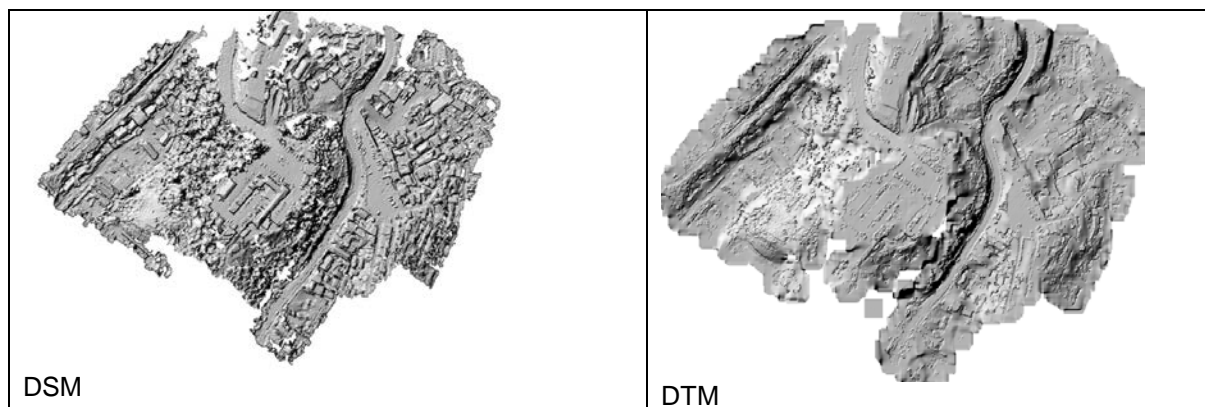
{
  "pipeline": [
    {
      "type": "readers.las",
      "filename": "wkhsourse2019.laz"
    },
    {
      "type": "filters.assign",
      "assignment": "Classification[:] = 0"
    },
    {
      "type": "filters.elm"
    },
    {
      "type": "filters.assign", "assignment": "NumberOfReturns[:] = 1"
    },
    {
      "type": "filters.assign", "assignment": "ReturnNumber[:] = 1"
    },
    {
      "type": "filters.outlier"
    },
    {
      "type": "filters.smrf",
      "ignore": "Classification[7:7]",
      "slope": 0.1,
      "window": 60,

```

```

    "threshold":0.4
  },
  {
    "type":"filters.range",
    "limits":"Classification[2:2]"
  },
  {
    {
      "type":"writers.gdal",
      "filename":" wkhsouce2019-dtm.tiff",
      "resolution":1,
      "output_type":"idw"
    }
  }
]
}

```






Generation of DTM from Point Clouds: A part in West Khasi Hills, Meghalaya

The above figure shows generation of DTM for a part of West Khasi Hills with modified parameters of SMRF ("type":"filters.smrf", "ignore":"Classification[7:7]", "slope":0.06,"window":49, "threshold":1.05) and written to a raster file using writers.gdal with parameters "resolution":0.1, "window_size":80, "output_type":"idw".

Learning Goal 10: Decimate Point Clouds for improve performance and reduce file size.

A Point Cloud can contain lacs of points. Storing, processing and handling such point clouds sometimes become troublesome. We can always decimate or discard repeated points without losing the information content. Decimation therefore can helps in improving performance and reducing disk storage. We below shows 2 ways of decimation with or without randomization. Create json files and copy the PDAL commands as listed below and run the pipeline.

Original Point Cloud	
<pre>{ "pipeline":["wkhsouce2019.las", { "type":"filters.decimation", "step": 20 }, { "type":"writers.las", "filename":"output_1.las" }] }</pre>	 <p>An output after applying decimation filter which picks every 20th point in an input las file</p>
<pre>{ "pipeline":["wkhsouce2019.las", { "type": "filters.randomize" }, { "type":"filters.decimation", "step": 20 }, { "type":"writers.las", "filename":"output_2.las" }] }</pre>	 <p>Adding a <i>filters.randomise</i> is a good step to spread the result output. Above shows the output where, we first randomises the order of points in the input data, then picks up every 20th point</p>

Conclusion

PDAL can be great for processing point cloud and derive useful results. For more better classification, **CGAL** may be used for properly identifying various objects and features on the scenes specially for Point Clouds derived via SfM approach. Python can be used as part of PDAL processing pipelines using filters.python.