Computers & Geosciences xxx (xxxx) xxx

Contents lists available at ScienceDirect

Computers and Geosciences

journal homepage: www.elsevier.com/locate/cageo

Research paper

PDAL: An open source library for the processing and analysis of point clouds

Howard Butler [a], Bradley Chambers [b], Preston Hartzell [c,*], Craig Glennie [c]

[a] *Hobu Inc, 103 East College St, Iowa City, IA 52240, USA*
[b] *Grover Consulting Services, 8006 Hedgewood Ct, Fairfax Station, VA 22039, USA*
[c] *University of Houston, 5000 Gulf Freeway, Houston, TX 77204, USA*

ARTICLE INFO

Keywords:
Point clouds
Lidar
Open source software
Geospatial
Iterative closest point

ABSTRACT

As large point cloud datasets become ubiquitous in the Earth science community, open source libraries and software dedicated to manipulating these data are valuable tools for geospatial scientists and practitioners. We highlight an open source library called the Point Data Abstraction Library, more commonly referred to by its acronym: PDAL. PDAL provides a standalone application for point cloud processing, a C++ library for development of new point cloud applications, and support for Python, MATLAB, Julia, and Java languages. Central to PDAL are the concepts of stages, which implement core capabilities for reading, writing, and filtering point cloud data, and pipelines, which are end-to-end workflows composed of sequential stages for transforming point clouds. We review the motivation for PDAL's genesis, describe its general structure and functionality, detail several options for conveniently accessing PDAL's functionality, and provide an example that uses PDAL's Python extension to estimate earthquake surface deformation from pre- and post-event airborne laser scanning point cloud data using an iterative closest point algorithm.

1. Introduction

The use of point clouds derived from lidar observations and photogrammetric methods has led to a revolution of fundamental discoveries in the Earth sciences (Glennie et al., 2013; Telling et al., 2017) in diverse fields such as snow depth estimation (Deems et al., 2013), active tectonics (Meigs, 2013), and the study of mass and energy transfer across landscapes (Passalacqua et al., 2015). According to Eitel et al. (2016), the number of studies included in the ISI Web of Science Core Collection containing the keywords lidar AND (earth OR ecology) has doubled since 2007, while the number of papers cited containing these keywords has increased six-fold. Many of the discoveries in these studies were based on interpretation and analysis of high-resolution digital elevation models (DEMs), i.e., 2.5D raster datasets, generated from the originally observed point cloud data. However, new discoveries increasingly benefit from direct analysis of the raw lidar point cloud observations, e.g., Ekhtari and Glennie (2018) and Scott et al. (2018), which require specialized tools for extraction, manipulation, and analysis of the unordered 3D point data.

Invariably, because of differences in data formats, resolution, classification schemes, or reference coordinate systems, individual geospatial point cloud datasets need a variety of pre-processing tasks applied to them before they are suitable for ingestion into point cloud analysis applications or custom algorithms. However, the large size of 3D point cloud datasets, typically numbering in millions to billions of points, requires the use of specialized, scalable algorithms in order to apply these initial processing steps in an efficient manner. Although the use of point cloud observations in the Earth sciences has increased dramatically, there remains a paucity of open source tools and algorithms specifically designed for geospatial point cloud pre-processing, filtering, and analysis.

Many open source libraries that support the management and processing of point cloud data do not incorporate geospatial reference frame and datum transformations or broad format input/output and translation capabilities. Instead, their focus is on core processing tasks often of interest to the computer science field such as feature segmentation and point clustering, cloud to cloud registration, mesh and raster creation, or 3D visualization. Examples include Open3D (Zhou et al., 2018), Cilantro (Zampogiannis et al., 2018), 3DTK (3DTK, 2020), and PCL (Rusu and Cousins, 2011). A popular point cloud software suite that does focus on the geospatial community is LAStools (Isenburg, 2020), which allows users to orchestrate point cloud data workflows through composition of individual command line utilities. However, LAStools utilizes a hybrid open source licensing scheme with some components requiring commercial licenses, and it is tuned for working with the American Society for Photogrammetry and Remote Sensing (ASPRS) LAS (LASer) data model (ASPRS, 2019).

In addition to open source libraries and command line tools, a number of open source graphical user interface software are also used

* Corresponding author.
E-mail address: pjhartzell@uh.edu (P. Hartzell).

https://doi.org/10.1016/j.cageo.2020.104680
Received 24 August 2020; Received in revised form 28 October 2020; Accepted 15 December 2020
Available online 24 December 2020

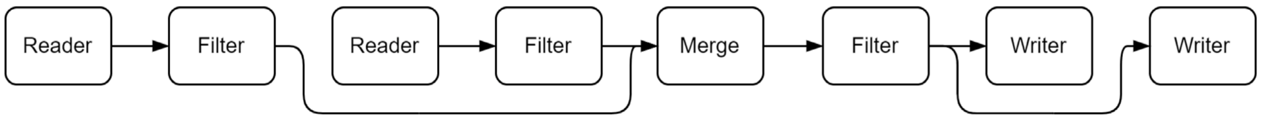

Fig. 1. General pipeline concept. Each box represents a sequential stage in the pipeline. Note that the "Merge" stage is categorized as a filter.

within the Earth science community, such as CloudCompare (CloudCompare, 2020) and BCAL (BCAL, 2020). However, these types of software packages exhibit a lack of extensibility and integration convenience to allow them to be easily placed in the context of larger software systems. Thus, there remains a significant need for flexible and extensible open source tools for use with large, high resolution 3D point cloud models in the Earth science fields.

The Point Data Abstraction Library (PDAL) fills this geospatial point cloud software niche by providing abstract access, filtering, exploitation, and workflow management capabilities in an open source package. PDAL is a C++ library with a JSON-based processing pipeline domain-specific language, Python, Java, Julia, and MATLAB support, and a convenient command line application. This multi-tiered approach allows a diverse audience of application developers, data processors, and scientists to use the capabilities of the library in familiar and convenient environments. In addition to handling over 30 data formats and robust geospatial reference frame and datum transformation capabilities, PDAL incorporates a broad array of point cloud processing options from basic ordering and culling to advanced feature generation, point clustering, registration, and meshing.

This article reviews the motivation for developing PDAL and its basic architecture, describes its extensive functionality, highlights convenient methods for using PDAL with a focus on PDAL's Python support, and provides a reference implementation of PDAL that measures the geospatial change that occurred due to the August 2014 Napa, California earthquake.

2. The PDAL library

PDAL evolved from the data management and translation activities centered around the libLAS library (Butler et al., 2019). The libLAS library addressed early challenges of using lidar data stored in the LAS format, chief of which was the lack of software packages or libraries that provided native LAS reading and writing abilities. Through development of libLAS – which is currently in maintenance-only mode – it became clear that a library designed for geospatial point cloud data processing and management was needed that could:

(1) *Utilize a declarative directed graph workflow syntax, i.e., a pipeline syntax,* for users to compose read, filter, and write steps into a single workflow activity.
(2) *Accommodate a dynamic data model*, where data can be composed of any user- or format-defined schema rather than fixed data arrangements.
(3) *Allow dynamic plugins*, both open source and proprietary, that use external libraries to extend the library's built-in read, filter, and write capabilities.
(4) *Provide an abstract C++ API for geospatial point cloud formats* focused on extract, transform, and load (ETL) operations that enables developers to directly integrate the library's functionality into their own C++ projects.

The initial 1.0.0 release of PDAL was made in September of 2015, but the library was used internally by the U.S. Army Corps of Engineers Cold Regions Research and Engineering Laboratory (CRREL) throughout its entire development period. The library is released at roughly semi-annual intervals, and each release brings new capabilities, additional documentation, and enhanced performance. Early releases of PDAL focused on data format translation with an emphasis on geospatial data types and enhanced with additional capabilities to crop, split, or transform points into different spatial reference systems. Since then, most development activity has focused on adding and enhancing filtering capabilities, with additional read and write format drivers added as needed.

PDAL has been released as open source software (BSD license) since its inception. Documentation and source code are available from the project website (https://pdal.io) with ongoing development coordinated via GitHub (https://github.com/PDAL/PDAL). PDAL binaries are available on all major operating systems via Conda (https://anaconda.org/conda-forge/pdal), Docker users can access PDAL via Docker Hub (https://hub.docker.com/r/pdal/pdal), and major Linux distributions such as RedHat and Debian include PDAL in their respective geographic information system (GIS) packaging sub-distributions.

2.1. Pipelines

PDAL's distinguishing design choice is to model the processing of point cloud data using the concept of a pipeline, or a directed graph. Point cloud data is read from a set of input sources using format-specific readers, the data is passed through various filters that transform data or create metadata, and the data is then written to an output stream using format-specific writers. The general concept of a pipeline is illustrated in Fig. 1. Each of the sequential actions, or processing modules, applied to a point cloud is referred to as a "stage" and falls into one of three categories: reader, filter, or writer.

Fig. 1 emphasizes the sequential nature of pipeline stage execution. Note that when a new reader stage is encountered, subsequent filter stages are applied only to the most recent data source. PDAL pipelines also support writing data to multiple output streams, i.e., multiple writer stages can be included in a pipeline. However, pipeline stages are always executed sequentially, never in parallel.

Stages can be composed into pipelines using a JSON array representation, with each stage name prefaced by its category. For example, a simple pipeline that reads a point cloud from a LAS format file (`readers.las`), applies a spatial filter to keep only those points inside specified X and Y coordinate ranges (`filters.crop`), and writes the cropped point cloud to a text format file (`writers.text`) can be expressed in JSON as:

```
[
    {
        "type":"readers.las",
        "filename":"input.las"
    },
    {
        "type":"filters.crop",
        "bounds":"([0,100],[0,100])"
    },
    {
        "type":"writers.text",
        "filename":"output.txt"
    }
]
```

Note that most stages have a number of options, some of which are required. For example, the `readers.las` stage requires the `filename` option and the `filters.crop` stage requires a `bounds` or polygon option. Significantly more complex and powerful pipelines than the example given above can be created (see https://pdal.io/pipeline.html for additional examples). However, the fundamental concept remains the same: *readers* provide data elements to the pipeline by reading one or more point cloud files, optional *filters* operate on the data as inline operations, and *writers* consume the data provided by the readers and write it to one or more data streams.

Pipelines saved in JSON format files can be conveniently executed using PDAL's command line application, pdal, which is reviewed in Section 4.1. Pipelines can also be executed with PDAL's Python and Java bindings, with the ability to apply further custom processing steps to the resultant point cloud using the facilities of those languages. A pipeline model provides several advantages over specific applications, e.g., command line tools that perform certain operations such as format translation or data reprojection. These advantages include:

(1) The pipeline serves as a record of the operations applied to the data.
(2) A skeleton of an operation can be constructed and specific options (e.g., filenames) substituted.
(3) Complex pipeline operations can be constructed using the JSON manipulation facilities in a user's chosen software language.

2.2. Data model

In order to be useful, all point cloud formats must contain data elements that provide some notion of spatial location (e.g., XYZ coordinates). Beyond location, however, the data housed in various formats may or may not have common data fields. Some formats predefine the data elements that make up a point, while other formats provide this information in a header or preamble. PDAL is format-agnostic, and therefore supplies a large selection of predefined data elements that are in common use by the formats that it currently supports. These data elements are referred to as "dimensions" in PDAL, where a dimension describes the combination of data type, size, and meaning. For example, the GpsTime dimension is a double data type, 64 bits in size, and represents the GPS time that a laser pulse was emitted by a lidar sensor.

PDAL currently supplies over 70 different dimensions. A complete listing can be found at https://pdal.io/dimensions.html. In addition to these predefined data dimensions, new dimensions with desired names and data types can be created by extending PDAL's C++ API, e.g., to support a custom data format reader that contains a data element not defined in the current set of dimensions. Information and examples on extending PDAL's C++ API are found on the project website (https://pdal.io).

2.3. Plugins

A plugin is a stage (a reader, writer, or filter) that has dependencies which are considered optional to the core PDAL project. Most of the known PDAL plugins (others could be proprietary) are packaged with PDAL and can optionally be built alongside PDAL when building the software from source. In most cases this allows the plugins to be tested with each PDAL build. Note that since plugins require external libraries, some plugins may not be included in the versions of PDAL installed via Conda. In these situations, users must obtain the required libraries and build PDAL from source on their local machine.

Three plugins that are openly published, but are external to PDAL, are: filters.align3d, writers.shr3d, and writers.prc. The first two were developed by the Johns Hopkins University Applied Physics Laboratory and can be found at https://github.com/pubgeo/pubgeo. The filters.align3d plugin is a data registration approach, while writers.shr3d produces bare earth digital elevation models (DEMs). The third external plugin, writers.prc, was created by PDAL contributors to generate PDF content with an embedded point cloud according to the PRC (Adobe Product Representation Compact) specification (ISO, 2014). It has been kept separate from the PDAL codebase due to licensing.

2.4. C++ API

PDAL provides a C++ API that can be used by programmers for integrating PDAL into their own projects or extending PDAL's existing capabilities. Documentation is supplied on PDAL's website at https://pdal.io/api/cpp/index.html, and the test suite and source code contained in PDAL's GitHub repository are recommended learning resources. In the interests of brevity and relevance to the widest audience, the C++ API is not detailed here. Rather, several convenient methods for accessing PDAL's functionality via PDAL's command line application and Python binding are detailed in Section 4.

3. Functionality

PDAL offers more than 100 reading, writing, and filtering stages. Rather than an exhaustive listing of the current stages (that would rapidly become outdated), we group the stages and provide general remarks about typical stage capabilities within each group. Documentation for all of PDAL's available stages is found on PDAL's website.

3.1. Readers and writers

Because of PDAL's history as an ETL data management tool focused on geospatial lidar, support for commonly encountered geospatial-centric formats in that domain is heavily emphasized. These include complete support for all point type variants of the ASPRS LAS format and its compressed counterpart, LASzip (Isenburg, 2013). Support for openly specified data formats such as BPF (NGA, 2005), PLY, PCD (PCD, 2020), TileDB (Papadopoulos et al., 2016) and text/CSV is also provided. Proprietary format support is provided by a number of binary software development kits and corresponding plugin implementations. These include database drivers for Oracle and PostgreSQL and binary formats such as Terrasolid, MrSID, and Riegl's RXP and RDB formats.

3.2. Filters

Filters can remove, modify, reorganize, and add points to the data stream as it is processed. PDAL filters also commonly create new dimensions or alter existing ones. For example, filters.hag_dem adds a HeightAboveGround dimension based on the height of a point above a DEM, and filters.smrf changes each point's Classification dimension value according to the determination by a simple morphological filter (Pingel et al., 2013) of whether a point lies on a ground or non-ground surface.

PDAL's broad array of filters can be grouped by their functional purpose:

- *Create:* Filters that create or alter dimensions (not points) as described above. This is a large category that includes functionality such as clustering, local covariance features, density computations, noise filtering, surface normal estimation, planar feature identification, and ground point classification.
- *Order:* Filters that change point order, e.g., ascending or descending order based on a given dimension or Morton ordering.
- *Move:* Filters that change point coordinates through geometric transformations, projection from one coordinate system to another, or via registration algorithms.
- *Cull:* Filters that remove points and return a point cloud smaller than the original. Examples include cropping points within a bounding box or polygon and Poisson sampling of a point cloud.
- *New:* Filters that split the incoming point cloud into subsets. This includes dividing points into equal sized groups or separating points based on lidar scan lines, among others.
- *Join:* A single filter that joins multiple point clouds (filters.merge) to form a single point cloud.

- *Metadata:* Filters that generate information about the point cloud, such as a point cloud's boundary or information such as point density, point count, spatial reference information, or dimension statistics.
- *Mesh:* Filters that generate a mesh representation using methods such as Delaunay triangulation or Poisson surface reconstruction.
- *Languages* - filters that embed software written in Python, MATLAB, or Julia as a stage in a pipeline.

4. Convenient access

PDAL's functionality can be conveniently accessed via its command line application and its support for the Python, Java, MATLAB, and Julia languages. PDAL provides extensions for Python and Java, where the extensions enable communication with PDAL inside Python or Java code. For example, the Python extension enables users to execute PDAL pipelines in Python and access the point cloud data via NumPy arrays. Support for MATLAB and Julia is limited to embedded stages, which enable processing algorithms written in these languages to be included as stages within PDAL pipelines. Embedded stages are also supported for Python. Additional details, including installation requirements, for PDAL's support of each language are provided on PDAL's website.

In the following, we review the command line application, which is included in all PDAL installations, and PDAL's support for Python, which is widely used in the Earth sciences community. PDAL's Python extension is also used in the example application given in Section 5.

4.1. Command line application

PDAL provides a single command line application called `pdal`. Operations are run by invoking the `pdal` application along with a command name. For example, information about a point cloud contained in a LAS file residing in the current directory can be displayed with the following command:

```
$ pdal info mypointcloud.las
```

where `pdal` is the name of the application, `info` is the command, and `mypointcloud.las` is the input to the command.

A number of common functions are available as distinct commands in the `pdal` application, such as format translation (`translate`), ground filtering (`ground`), and point cloud merging (`merge`), splitting (`split`), and sorting (`sort`) abilities. However, the entire functionality of the PDAL library is available through the `pipeline` command, with the exception of plugins that a user has chosen not to build on their local machine. The `pipeline` command accepts a JSON file containing the desired stages in the format described in Section 2.1:

```
$ pdal pipeline mypipeline.json
```

It is also possible to apply a filters-only pipeline (no reader or writer stages are specified in the JSON pipeline file, only filter stages) in a call to `pdal translate` with the `--json` switch, e.g.,

```
$ pdal translate input.las output.las --json myfilters.json
```

This is useful to quickly apply a series of filters to multiple point cloud files without having to repeatedly specify the input and output files in the JSON pipeline file. In this case, PDAL infers the appropriate reader and writer from the passed file types.

Help for the application or an individual command can be retrieved by appending the `--help` switch. The `--drivers` switch will list all available reader, filter, and writer stages, and the `--options` switch followed by a stage name will provide information about that stage and its options, e.g.,

```
$ pdal --options readers.las
```

4.2. Python

PDAL provides Python support (Python 3.6+) in two ways. First, it embeds Python to allow a user to write Python programs that interact with data using the `filters.python` stage. Second, it extends Python by providing an extension that Python programmers can import to leverage PDAL capabilities in their own applications. Both of these capabilities are provided in the `python-pdal` Conda package. Since the Python capabilities require a PDAL installation, installing the `python-pdal` Conda package also provides a complete PDAL installation.

Embedding Python refers to a user inserting Python functions inline with other stages via `filters.python` in a PDAL pipeline, enabling modification of PDAL points through a NumPy array. The Python function must have two NumPy arrays as arguments: `ins` and `outs`. The `ins` array represents the points before the `filters.python` filter and the `outs` array represents the points after filtering. A simple example Python function that scales the Z coordinate of each point by a factor of ten is:

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

Note that the Python function used in `filters.python` must always return `True` upon success. A corresponding JSON pipeline that contains a `filters.python` stage implementing the `multiply_z` Python function could be:

```json
[
    {
        "type":"readers.las",
        "filename":"input.las"
    },
    {
        "type":"filters.python",
        "script":"multiply_z.py",
        "function":"multiply_z"
    },
    {
        "type":"writers.las",
        "filename":"output.las"
    }
]
```

The purpose of the embedded Python language stage is to allow users to write small programs that implement interesting actions without requiring the full C++ development activity of building a PDAL stage. A Python filter is an opportunity to interactively and iteratively prototype a data operation without strong considerations of performance or generality. This applies to MATLAB and Julia embedded stages as well. Once the prototype filter is operating as desired, a user could optionally formalize the filter in C++ as a custom PDAL stage.

In contrast to the embedded stage functionality described above, PDAL's Python extension enables users to execute pipelines from within Python and capture the results as NumPy arrays. This mode of operation is useful to users interested in having PDAL simply act as a data format and processing handler. Python extension users are expected to construct their own PDAL pipeline in text form or by using Python's `json` library or other library of their choice for manipulating JSON. The JSON pipeline is then fed into the extension and NumPy arrays are returned. A simple example where the Python extension is used to read a LAS format point cloud and sort by the X coordinate, with the resulting NumPy data exposed to the user, is:

```
json_pipeline = """
[
    {
        "type":"readers.las",
        "filename":"input.las"
    },
    {
        "type":"filters.sort",
        "dimension":"X"
    }
]
"""


import pdal
pipeline = pdal.Pipeline(json_pipeline)
pipeline.validate()
pipeline.execute()
arrays = pipeline.arrays
array = arrays[0]
x = array['X']
y = array['Y']
z = array['Z']
```

Additional dimensions are accessed in the same way as the XYZ dimensions shown in the example code above. Note that it is necessary to provide an array index (`array = arrays[0]` line in the above code listing) since it is possible that more than one array could be available, e.g., in the presence of multiple reader stages or splitting filters.

By embedding and extending the Python language, PDAL's point cloud manipulation and processing services are accessible to a broad user base in the scientific community. The example application in the following section further illustrates the use of PDAL's Python extension and command line application for a geospatial change detection analysis in the Earth sciences field.

5. Example application

Within the applications of point clouds in the Earth sciences, an emerging use of the data is for change detection, i.e., the comparison of temporally spaced 3D models for determination of landscape evolution. Change detection has been especially present in the field of active tectonics, where pre- and post-event point clouds are examined to determine the amount of surface deformation caused by an earthquake, see, for example, Oskin et al. (2012), Nissen et al. (2014), Ekhtari and Glennie (2018) and Scott et al. (2018). The majority of these tectonic studies make use of the iterative closest point (ICP) algorithm (Besl and McKay, 1992) for estimation of the 3D deformation field.

Motivated by the prevalence of the ICP algorithm in Earth science change detection, we present a reference implementation using PDAL's `filters.icp` stage to estimate surface deformation caused by an earthquake. The example will utilize pre- and post-event airborne lidar data of the M6.0 Napa, California earthquake, which occurred on 24 August 2014. The pre-event data was collected with a nominal point density of 8 pts/m^2 in June 2014 under contract with the city of Napa for the purpose of updating existing city maps. The post-event data was collected by Towill Inc. for the USGS in September 2014 with a slightly higher point density of 11 pts/m^2. Additional details on the pre- and post-event lidar data can be found in Ekhtari and Glennie (2018), while details regarding the Napa earthquake are given in Brocher et al. (2015). The analysis area is shown in Fig. 2.

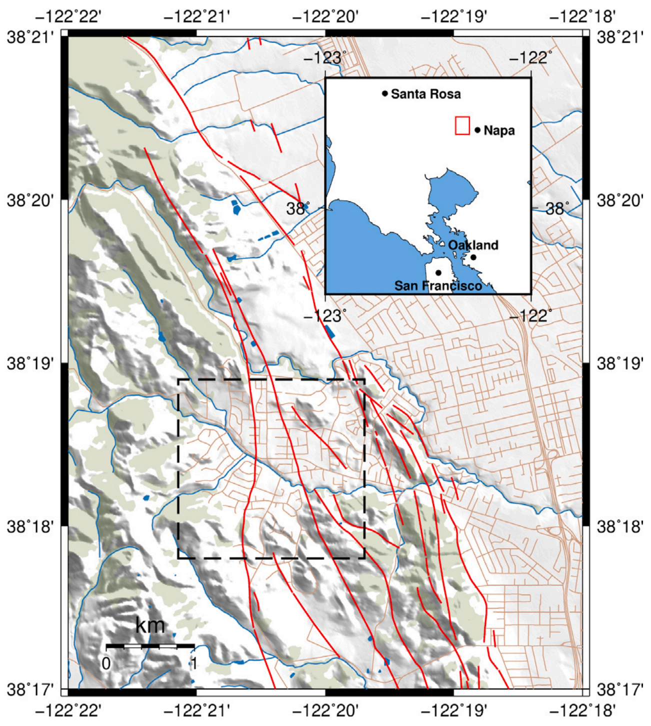

Fig. 2. Overview of fault locations (red lines) in the Brown's Valley area east of Napa. The black dashed box indicates the lidar data extent used in the analysis.

5.1. *Point cloud preparation*

Standard ICP algorithms are best applied to point clouds that contain only hard surfaces, i.e., when vegetation has been removed (Ekhtari and Glennie, 2018). Three PDAL pipelines were therefore applied to the pre- and post-event lidar point clouds to prepare them for the ICP algorithm:

(1) A ground filtering pipeline.
(2) A pipeline to identify points on building surfaces (primarily rooftops) by extracting planes from the non-ground points.
(3) A pipeline to crop and merge the identified ground and planar surface points.

Explanatory comments for each stage of the three pipelines are given in the following three subsections. The JSON pipelines are available on GitHub (see Section 7).

5.1.1. *Ground filter pipeline*

(1) `readers.las`. Open the original point file.
(2) `filters.assign`. Assign all point classification codes to 0 to remove their original values.
(3) `filters.elm`. Apply the extended local minimum (ELM) filter (Chen et al., 2012) to identify below-ground outliers in preparation for ground filtering with a simple morphological filter (Pingel et al., 2013), which is sensitive to low points. The ELM filter assigns a classification code of 7 to the identified noise points in accordance with the ASPRS LAS specification.
(4) `filters.smrf`. Identify ground points with the simple morphological filter. The identified ground points are assigned a classification code of 2 by the filter, and all previously identified noise points are explicitly ignored by the filter through use of the `ignore` option.
(5) `filters.outlier`. Apply an outlier filter, tuned with the `multiplier` option, to identify spurious in-air points. As with

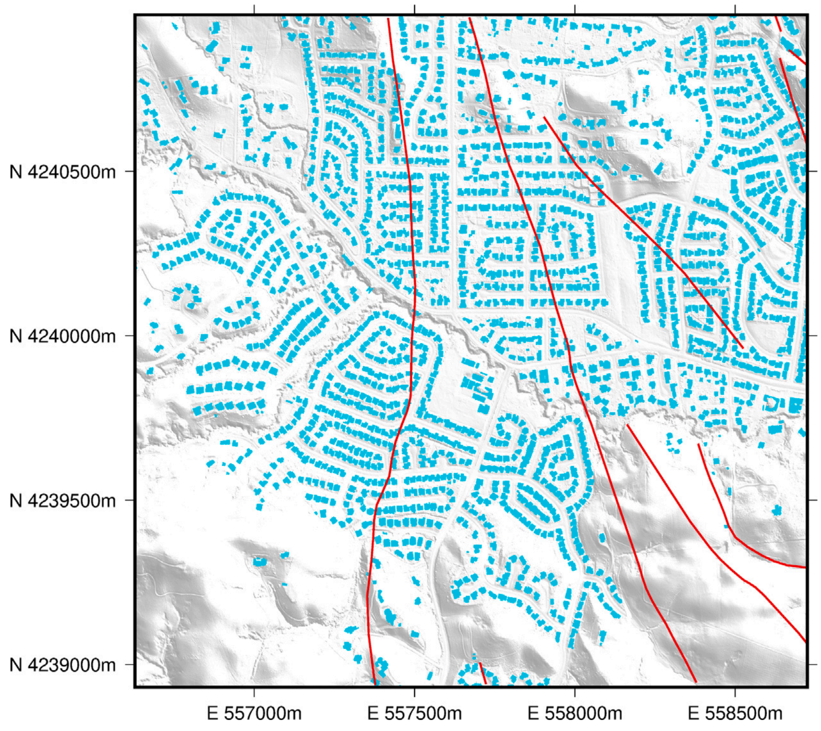

Fig. 3. DEM generated from filtered ground points of the post-event point cloud. Filtered planar points shown in cyan. Fault traces shown in red. Coordinates are Universal Transverse Mercator, Zone 11.

the ELM filter, the identified noise points are assigned a classification code of 7. This stage could optionally be applied prior to the SMRF stage or applied at a later location within the subsequent pipelines.

(6) `writers.las`. Save all points to file with their updated classification codes.

5.1.2. *Plane filter pipeline*

(1) `readers.las`. Open the ground filtered point file generated by the prior pipeline.
(2) `filters.sample`. Remove lidar scanning patterns and approximately match the pre- and post-event point densities by sampling the point cloud with a minimum point to point radius (`radius` option).
(3) `filters.hag_nn`. Estimate the height above ground (HAG) of each point using the nearest classified ground point as reference. The stage stores the HAG values in a new dimension named `HeightAboveGround`.
(4) `filters.range`. Retain only those points with HAG values where rooftops are reasonably expected, 2–20 m above ground in this case.
(5) `filters.approximatecoplanar`. Identify points in locally planar areas based on eigenvalues computed from the local neighborhood of points. The stage stores a value of 0 (non-coplanar) or 1 (coplanar) for each point in a new dimension named `Coplanar`. Note that `filters.covariancefeatures` provides several local geometry descriptors, one of which is a planarity feature that could be used here as well.
(6) `filters.range`. Retain only those points identified as coplanar.

We note here that the remaining coplanar points are heavily populated with vegetation points that randomly happen to lie within a plane. Most of these vegetation points could be removed by increasing the number of neighboring points (`knn` option) and modifying the eigenvalue ratios (`thresh1` and `thresh2` options) in the `filters.approximatecoplanar` stage. However, making these changes also reduces the number of rooftop points identified as coplanar. Instead, we take advantage of the relatively sparse locations of the remaining vegetation points in the following stage.

(7) `filters.cluster`. Assign unique IDs to point clusters defined by a maximum point to point distance (`tolerance` option) and a minimum population (`min_points` option). The stage stores the cluster IDs in a new dimension named `ClusterID`.
(8) `filters.range`. Retain only those points having a non-zero cluster ID. This eliminates the sparse vegetation points, which are not identified as clustered.
(9) `filters.assign`. Assign the remaining points a classification code of 6 in accordance with the ASPRS LAS specification for building points.
(10) `writers.las`. Save the planar points to file.

5.1.3. *Merge pipeline*

(1) `readers.las`. Open the point file generated by the ground filter pipeline.
(2) `filters.range`. Retain only those points classified as ground.
(3) `filters.sample`. As with the prior pipeline, remove lidar scanning patterns and match the pre- and post-event point densities by sampling the retained ground points with a minimum point to point radius (`radius` option).
(4) `filters.crop`. Crop point cloud to area common to both the pre- and post-event data.
(5) `readers.las`. Open the point cloud containing the coplanar points that we would like to merge with the ground points.

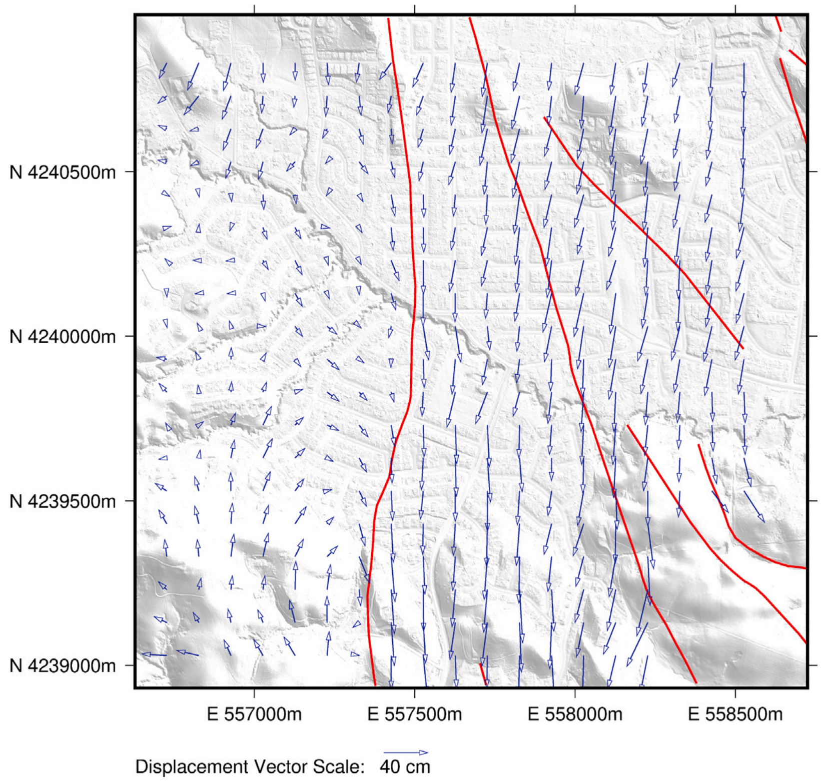

Fig. 4. ICP displacement vectors overlaid on a DEM generated from the filtered ground points of the post-event point cloud. The omitted displacement vectors in the southeast corner coincide with a void in the pre-event lidar point cloud. Fault traces shown in red. Coordinates are Universal Transverse Mercator, Zone 11.

(6) `filters.crop`. Crop point cloud to area common to both the pre- and post-event data.
(7) `filters.merge`. Merge the ground and coplanar point clouds.
(8) `writers.las`. Save the merged point cloud to file.

Note that once the second `readers.las` stage (stage #5) is executed, the subsequent `filters.crop` stage is only applied to this second set of data (see Fig. 1 for a conceptual outline of a pipeline that illustrates this characteristic). The prepared post-event data is visualized in Fig. 3 along with the fault lines.

5.2. Point cloud indexing

In order to capture the non-rigid deformation caused by the earthquake, we follow the approach in Ekhtari and Glennie (2018) and repeatedly apply ICP to the filtered pre- and post-event points falling within a sliding window. This requires a mechanism for repeatedly querying the pre- and post-event points clouds for data contained within the moving (sliding) window. Although PDAL's `filters.crop` stage could be used for this, it requires the pre- and post-event point clouds to be loaded into memory for each instance of the `filters.crop` stage, which is required for each window location. This repetitive file loading slows the process, and a more efficient alternative is desirable.

Indexing a point cloud into Entwine (see https://entwine.io/) Point Tiles (EPT) enables greater efficiency when repetitively querying a point cloud for data within a spatial boundary or for reduced resolution sampling. Once an EPT point cloud index has been created, PDAL's `readers.ept` stage can be used to extract point cloud data lying within spatial bounds designated by the user. Similar to PDAL, the Entwine library can be installed via Conda, and building an EPT index is a one line command in a terminal. For example, the command:

```
$ entwine build -i ./MyPointCloud.laz -o ./MyIndex
```

generates an EPT index from `MyPointCloud.laz` and stores it in the `MyIndex` directory. For this example application, unique EPT indices are created for the prepared pre- and post-event point clouds.

Note that methods faster than using EPT indices exist for accessing full resolution point data withing a spatial boundary, e.g., indexed tiles of full resolution point data. We have used EPT indices here for implementation simplicity (a single command creates an index that can be read by PDAL) while still gaining faster access to the windowed point data.

5.3. ICP with PDAL's Python extension

The ICP algorithm is applied to each window location using a pipeline consisting of a `readers.ept` stage for the post-event point cloud index, a `readers.ept` stage for the pre-event point cloud index, and a `filters.icp` stage. Since the `bounds` options of the `readers.ept` stages must be redefined for each sliding window location and a convenient mechanism for storing the computed displacement information is desirable, the ICP algorithm is implemented in a Python script using PDAL's Python extension. A nested loop structure is used to slide a square window through the analysis area in the X and Y directions. A number of items are implemented within the loop: the pipeline JSON text is created with the current window location defined in the `bounds` options of the `readers.ept` stages, the pipeline is executed, and the solved ICP translation components are extracted from the metadata returned by PDAL and stored for later export and plotting. For this example, a 200 m square window was used with a 100 m step between successive window locations. The complete Python script used to generate the ICP vectors is available on GitHub (see Section 7).

The horizontal components of the displacement vectors generated by the Python script are shown in Fig. 4. Note that, according to field observations, only the westernmost fault line ruptured during the Napa Valley earthquake, with a right lateral strike slip of approximately 30 to 40 cm (Brocher et al., 2015). The relative motion along the west fault line is readily apparent in the pattern of vectors in Fig. 4, and agrees both in magnitude and orientation to the field observations of displacement and the independent ICP displacement estimates presented in Ekhtari and Glennie (2018).

6. Conclusions

The open-source Point Data Abstraction Library has been briefly presented in this manuscript, including the motivation for its genesis, methods of use and access, and its extensive functionality. The library supports the needs of the Earth science community for a free and open-source solution for processing large point cloud datasets that is format agnostic and focused on geospatial translation and transformation. In addition to these core abilities, PDAL provides advanced processing algorithms for feature extraction, ground filtering, registration, meshing, and more. The library is extensible by software developers via its C++ API, and its functionality is easily accessed by a broad spectrum of users via its command line application and support for the Python, Java, MATLAB, and Julia languages. This access was illustrated with an example application that measured earthquake ground motion by way of several point cloud processing pipelines executed with PDAL's command line application, followed by application of PDAL's ICP filter using the Python extension.

PDAL continues to be actively developed, with future effort focusing on performance and efficiency enhancements for baseline operations, improved mesh and triangulation format support, and alternative high-level C++ APIs. Future releases will also include new feature extraction filters that will augment PDAL's existing filter suite to support machine learning applications that operate on 3D point cloud data. Going forward, PDAL will continue to support open-source integration with other toolkits and PDAL API consumers, e.g., the recent initiative to integrate point cloud visualization into QGIS, a popular open-source GIS software.

7. Computer code availability

PDAL is released under the BSD license. Source code is available at https://github.com/PDAL/PDAL. PDAL binaries are available on all major operating systems via Conda at https://anaconda.org/conda-forge/pdal. Docker users can access PDAL via Docker Hub at https://hub.docker.com/r/pdal/pdal.

Pipelines and Python script for the example application were written by Preston Hartzell and are available at: https://github.com/pjhartzell/pdal-icp-example.

CRediT authorship contribution statement

Howard Butler: Conceptualization, Software, Funding acquisition, Project administration, Writing - review & editing. **Bradley Chambers:** Software, Writing - original draft, Writing - review & editing. **Preston Hartzell:** Investigation, Writing - original draft, Writing - review & editing. **Craig Glennie:** Funding acquisition, Writing - original draft, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the U.S. Army Corps of Engineers, Cold Regions Research and Engineering Laboratory and the National Science Foundation, USA (grant numbers 1347092, 1830734).

References

3DTK, 2020. 3DTK - the 3D toolkit. http://slam6d.sourceforge.net/, (Accessed: 01 August 2020).
ASPRS, 2019. LAS specification VERSION 1.4 – R14. pp. 1–28, http://www.asprs.org/wp-content/uploads/2019/03/LAS_1_4_r14.pdf, (Accessed: 01 August 2020).
BCAL, 2020. BCAL lidar tools. https://www.boisestate.edu/bcal/tools-resources/bcal-lidar-tools/, (Accessed: 01 August 2020).
Besl, P., McKay, N.D., 1992. A method for registration of 3-D shapes. IEEE Trans. Pattern Anal. Mach. Intell. 14 (2), 239–256. http://dx.doi.org/10.1109/34.121791.
Brocher, T.M., Baltay, A.S., Hardebeck, J.L., Pollitz, F.F., Murray, J.R., Llenos, A.L., Schwartz, D.P., Blair, J.L., Ponti, D.J., Lienkaemper, J.J., Langenheim, V.E., Dawson, T.E., Hudnut, K.W., Shelly, D.R., Dreger, D.S., Boatwright, J., Aagaard, B.T., Wald, D.J., Allen, R.M., Barnhart, W.D., Knudsen, K.L., Brooks, B.A., Scharer, K.M., 2015. The Mw6.0 24 August 2014 South Napa earthquake. Seismol. Res. Letters 86 (2A), 309–326. http://dx.doi.org/10.1785/0220150004.
Butler, H., Łoskot, M., et al., 2019. LibLAS LAS 1.0/1.1/1.2 ASPRS LiDAR data translation toolset. https://liblas.org, (Accessed: 30 March 2019).
Chen, Z., Devereux, B., Gao, B., Amable, G., 2012. Upward-fusion urban DTM generating method using airborne Lidar data. ISPRS J. Photogrammetry Remote Sens. 72, 121–130. http://dx.doi.org/10.1016/j.isprsjprs.2012.07.001.
CloudCompare, 2020. Cloudcompare: 3D point cloud and mesh processing software. http://www.cloudcompare.org/, (Accessed: 01 August 2020).
Deems, J.S., Painter, T.H., Finnegan, D.C., 2013. Lidar measurement of snow depth: a review. J. Glaciol. 59 (215), 467–479. http://dx.doi.org/10.3189/2013JoG12J154.
Eitel, J.U., Höfle, B., Vierling, L.A., Abellán, A., Asner, G.P., Deems, J.S., Glennie, C.L., Joerg, P.C., LeWinter, A.L., Magney, T.S., Mandlburger, G., Morton, D.C., Müller, J., Vierling, K.T., 2016. Beyond 3-D: The new spectrum of lidar applications for earth and ecological sciences. Remote Sens. Environ. 186, 372–392. http://dx.doi.org/10.1016/j.rse.2016.08.018.
Ekhtari, N., Glennie, C., 2018. High-resolution mapping of near-field deformation with airborne earth observation data, a comparison study. IEEE Trans. Geosci. Remote Sens. 56 (3), 1598–1614. http://dx.doi.org/10.1109/TGRS.2017.2765601.
Glennie, C.L., Carter, W.E., Shrestha, R.L., Dietrich, W.E., 2013. Geodetic imaging with airborne LiDAR: the Earth's surface revealed. Rep. Progr. Phys. 76 (8), 086801. http://dx.doi.org/10.1088/0034-4885/76/8/086801.
Isenburg, M., 2013. LASzip: lossless compression of LiDAR data. Photogramm. Eng. Remote Sens. 79, http://dx.doi.org/10.14358/PERS.79.2.209.
Isenburg, M., 2020. LAStools - efficient LiDAR processing software. http://rapidlasso.com/LAStools, Accessed: 2020-08-04.
ISO, 2014. Document Management—3D use of Product Representation Compact (PRC) format—Part 1: PRC 10001. Standard, (ISO 14739-1:2014(E)), International Organization for Standardization, Geneva, Switzerland.
Meigs, A., 2013. Active tectonics and the LiDAR revolution. Lithosphere 5 (2), 226–229. http://dx.doi.org/10.1130/RF.L004.1.
NGA, 2005. NGA.SIG.0020_1.0_BPF: Binary point file 3 (BPF3) BPF public license file format definition. https://nsgreg.nga.mil/doc/view?i=4202, Accessed: 2019-05-31.
Nissen, E., Maruyama, T., Arrowsmith, J.R., Elliott, J.R., Krishnan, A.K., Oskin, M.E., Saripalli, S., 2014. Coseismic fault zone deformation revealed with differential lidar: Examples from Japanese Mw ~7 intraplate earthquakes. Earth Planet. Sci. Lett. 405, 244–256. http://dx.doi.org/10.1016/j.epsl.2014.08.031.
Oskin, M.E., Arrowsmith, J.R., Corona, A.H., Elliott, A.J., Fletcher, J.M., Fielding, E.J., Gold, P.O., Garcia, J.J.G., Hudnut, K.W., Liu-Zeng, J., Teran, O.J., 2012. Near-field deformation from the El Mayor–Cucapah earthquake revealed by differential LIDAR. Science 335 (6069), 702–705. http://dx.doi.org/10.1126/science.1213778.
Papadopoulos, S., Datta, K., Madden, S., Mattson, T., 2016. The tiledb array data storage manager. Proc. VLDB Endow. 10 (4), 349–360. http://dx.doi.org/10.14778/3025111.3025117.
Passalacqua, P., Belmont, P., Staley, D.M., Simley, J.D., Arrowsmith, J.R., Bode, C.A., Crosby, C., DeLong, S.B., Glenn, N.F., Kelly, S.A., Lague, D., Sangireddy, H., Schaffrath, K., Tarboton, D.G., Wasklewicz, T., Wheaton, J.M., 2015. Analyzing high resolution topography for advancing the understanding of mass and energy transfer through landscapes: A review. Earth-Sci. Rev. 148, 174–193. http://dx.doi.org/10.1016/j.earscirev.2015.05.012.
PCD, 2020. The PCD (Point Cloud Data) file format. https://pcl.readthedocs.io/projects/tutorials/en/latest/pcd_file_format.html#pcd-file-format, Accessed: 2020-08-18.
Pingel, T., Clarke, K., Mcbride, W., 2013. An improved simple morphological filter for the terrain classification of airborne LIDAR data. ISPRS J. Photogramm. Remote Sens. 77, 21–30. http://dx.doi.org/10.1016/j.isprsjprs.2012.12.002.
Rusu, R.B., Cousins, S., 2011. 3D is here: Point cloud library (PCL). In: 2011 IEEE International Conference on Robotics and Automation. IEEE, pp. 1–4. http://dx.doi.org/10.1109/ICRA.2011.5980567.

Scott, C., Arrowsmith, J., Nissen, E., Lajoie, L., Maruyama, T., Chiba, T., 2018. The M7 2016 kumamoto, Japan, earthquake: 3-d deformation along the fault and within the damage zone constrained from differential lidar topography. Journal of Geophysical Research: Solid Earth 123 (7), 6138–6155. http://dx.doi.org/10.1029/2018JB015581.

Telling, J., Lyda, A., Hartzell, P., Glennie, C., 2017. Review of Earth science research using terrestrial laser scanning. Earth-Sci. Rev. 169, 35–68. http://dx.doi.org/10.1016/j.earscirev.2017.04.007.

Zampogiannis, K., Fermuller, C., Aloimonos, Y., 2018. cilantro: A lean, versatile, and efficient library for point cloud data processing. In: Proceedings of the 26th ACM International Conference on Multimedia. MM '18, ACM, New York, NY, USA, pp. 1364–1367. http://dx.doi.org/10.1145/3240508.3243655.

Zhou, Q.-Y., Park, J., Koltun, V., 2018. Open3D: A modern library for 3D data processing. ArXiv abs/1801.09847.