3D-tools Documentation

Release 0.2.1

Zac Hatfield Dodds

2017-04-07

Contents

`3D-tools` is a set of tools for analysing pointclouds of a forest.

`main.py` can be used by developers, but for general use you can instead `pip install forestutils`, which makes the `forestutils` command available from your terminal.

Features:

- Detect, map, and extract trees from the full pointcloud
- Calculate location, height, canopy area, and colour of each tree
- Losslessly reduce pointcloud size by discarding ground points

It is written in pure Python (3.4+), available under the GPL3 license, and can analyse multi-gigabyte datasets in surprisingly little memory.

CHAPTER 1

User Documentation

Contributing to forestutils

The `forestutils` package is licensed under the GNU General Public License, version 3 or later.

Contributions under that license are welcome; a pull request on GitHub is the preferred format.

If possible, please ensure that the test suite passes - see `.travis.yml`, and get the test dependancies with `pip install forestutils[test]`.

Non-code contributions, such as improved documentation or bug reports, are also welcome.

Collecting Input Data

Wanted: XYZRGB Pointclouds

In `.ply` format. See *usage tips* for more details.

LIDAR

If you have access to LIDAR hardware expensive enough to do full-colour scans, you shouldn't need my help operating it. Have fun, fortunate user!

Photogrammetry

Photogrammetry, sometimes called Structure-from-Motion (SFM) is the technique of reconstructing a 3D scene from multiple 2D images.

With a quadcopter to make landscape-scale scanning practical, this is a viable and reasonably-priced option for research, forestry, and citizen forest monitoring programmes.

The tutorials on flightriot.com are a good overview of the whole process.

Flight paths

The key property of a flight path for photogrammetry is that every part of the scene must be visible in multiple images, from multiple angles. Images should have a roughly one-third overlap in each direction.

Note that while it is tempting to use a fisheye lens for a wider field of view, the avoiding image distortion is much more important. The less reprojection must be done in software, the better! Also ensure that the shutter speed is fast enough to avoid motion blur, and that geolocation metadata is of reasonably high precision.

- The classic option is a simple grid pattern with the camera facing directly downwards. This generally works well at low altitude; coverage of horizontal surfaces is good but the sides of trees are often missed.
- Grid patterns with an angled camera (eg 45 degrees) work very well, but require many more photographs - essentially flying the pattern multiple times in different directions. One is similar to a nadir grid; two would be better, and I suspect three would give very good results.
- An 'orbit' pattern works well if only a single landscape feature is of much interest. Ensure a variety of angles (including altitude) and distances - without this the reconstruction will likely fail.

Software options

There is a wide variety of photogrammetry software out there; this section only mentions those useful at landscape-scale. Feel free to contribute examples!

I do not know of any photogrammetry packages with a license that meets the Open Source definition, which is frustrating for would-be developers and non-research users. I encourage anyone interested to start or support a project to fill this gap.

Software	License	Comments
VisualSFM	Free for noncommercial use, unmodified redistribution	Works well with limited compute power available; low cost.
CMP-MVS	Research use only (free).	Requires unusually powerful hardware (eg 4GB+ video memory).
Pix4D	Sells expensive commercial and costly educational licenses.	Very expensive, but excellent results. Trial version works, except exporting files!

Suggested settings

The various software options are quite distinct, but a few tips carry over.

- Performance

  3D reconstruction is more computationally intensive than any common desktop application. Look at all the options; some shave hours off the running time.

  - Ensure that GPU acceleration is enabled
  - Test a subset of your data, and run the full set overnight
  - Reserve 1GB RAM and a CPU core, if you want to run anything else

- Completeness of reconstruction

  Trees look very different from slightly different angles, so some settings tweaks are useful to ensure they are in the scene as well as the ground. Remember that `forestutils` will drop spurious ground points later.

- Generally a point is constructed if it is recognised in 3 images. Turn this down to 2 images.
- Use reduced image scale for the keypoint reconstruction stage.
- Use full image scale (or even more) for point densification.
- Turn desired point density to maximum.

Using forestutils

Installation

You will need Python *installed, version 3.4 or later.*

Run `pip install forestutils`. That's it! You can now run `forestutils` from the commandline without any further configuration.

Command-line usage

See the output of `forestutils --help` for detailed information on command arguments.

To summarise: you supply an input filename, of a pointcloud in .ply_ format. Optional arguments include output directory, whether to save each tree to a seperate file, and so on.

However, not every valid `.ply` file is a valid input for `forestutils` - in practise, the format is as flexible as `.csv` and the parser sacrifices some flexibility for speed.

- If you are using pointclouds generated by Pix4D, no changes are required. For multi-part pointclouds you can pass the name of `*_part_1.ply` and it will be treated as a single file (ensure the `part_N_ply_offset.xyz` files are present for this).
- For other pointcloud formats, simply open and resave in MeshLab. This allows forestutils to defer the problem to a more robust program. It also confers second-hand 'compatibility' with a wider variety of 3D formats!
- For LIDAR outputs, `.las` format is very common but unfortunately not supported at this time. (If you have LIDAR you can probably afford to convert data formats though)

Any pointcloud output by forestutils is of course also a valid input, and repeated processing (including size-reduction) should have a limited impact on data completeness (tests show a 1-5% loss in some circumstances).

CHAPTER 2

Code Documentation

- genindex

Changelog

Please note that `forestutils` is considered to be in alpha. Semantic versions below `1.x.y` indicate that anything may change without notice.

```
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
```

0.2.0

Upgraded to handle pointclouds with any vertex attributes - no longer just XYZRGB. Useful for FLIR, multispectral, or even hyperspectral data!

Consequential changes also made .ply parsing more robust towards unexpected data or unknown formats.

More tests again, adding type analysis with `mypy`.

0.1.2

Improved testing structure and declared dependencies for testing.

0.1.1

Improved documentation.

0.1.0

First public release on PyPi.

Earlier versions were developed for a remote sensing project, and internal use in the Borevitz Lab.

forestutils

geoply

pointcloudfile

A specialised io module for binary `.ply` files containing XYZRGB points.

Most uses of this module should go through *`read()`* to iterate over points in the file, or *`write()`* to save an iterable of points. Neither function accumulates much data in memory.

`IncrementalWriter` is useful when accumulating data in memory to write many files is impractical. *`offset_for()`* and `read_header()` provide location metadata if possible.

In all cases a "point" is tuple of (x, y, z, r, g, b). XYZ are floats denoting spatial coordinates. RGB is the color, each an unsigned 8-bit integer. While intentionally limited in scope, most data can be converted to this format easily enough.

class `src.pointcloudfile.`**`IncrementalWriter`**(*filename: str*, *header: src.pointcloudfile.PlyHeader*, *utm: src.pointcloudfile.UTM_Coord = None*, *buffer=4194304*) → None

A streaming file writer for point clouds.

Using the IncrementalWriter with spooled temporary files, which are only flushed to disk if they go above the given size, allows for streaming points to disk even when the header is unknown in advance. This allows some nice tricks, including splitting a point cloud into multiple files in a single pass, without memory issues.

Parameters

- **`filename`** – final place to save the file on disk.
- **`source_fname`** – source file for the pointcloud; used to detect file format for metadata etc.
- **`buffer`** (*`int`*) – The number of bytes to hold in RAM before flushing the temporary file to disk. Default 1MB, which holds ~8300 points - enough for most objects but still practical to hold thousands in memory. Set a smaller buffer for large forests.

class `src.pointcloudfile.`**`PlyHeader`**(*vertex_count*, *names*, *form_str*, *comments*)

Create new instance of PlyHeader(vertex_count, names, form_str, comments)

`comments`
Alias for field number 3

`form_str`
Alias for field number 2

`names`
Alias for field number 1

`vertex_count`
Alias for field number 0

class src.pointcloudfile.**UTM_Coord**(*x*, *y*, *zone*, *north*)
Create new instance of UTM_Coord(x, y, zone, north)

north
Alias for field number 3

x
Alias for field number 0

y
Alias for field number 1

zone
Alias for field number 2

src.pointcloudfile.**offset_for**(*filename: str*) → typing.Tuple[float, float, float]
Return the (x, y, z) UTM offset for a Pix4D or forestutils .ply file.

src.pointcloudfile.**parse_ply_header**(*header_text: bytes*) → src.pointcloudfile.PlyHeader
Parse the bytes of a .ply header to useful data about the vertices.

Deliberately discards the non-vertex data - this is a pointcloud module!

src.pointcloudfile.**ply_header_text**(*filename: str*) → bytes
Return the exact text of the header of the given .ply file, as bytes.

Using bytes to allow len(header) to give index to start of data; it's trivial to decode in the parsing function.

src.pointcloudfile.**read**(*fname: str*) → typing.Iterator
Passes the file to a read function for that format.

src.pointcloudfile.**write**(*cloud: typing.Iterator*, *fname: str*, *header: src.pointcloudfile.PlyHeader*, *utm: src.pointcloudfile.UTM_Coord*) → None
Write the given cloud to disk.

CHAPTER 3

Useful links

- documentation
- processed pointclouds
- maps (from output)
- source code
- forestutils package on PyPi

Contact: *zac.hatfield.dodds@anu.edu.au*

Python Module Index

S

Index