pyfor Documentation

Bryce Frank

Dec 01, 2019

Contents

pyfor is a Python package for processing and manipulating point cloud data for analysis in large scale forest inventory systems. pyfor is developed with the philsophy of flexibility, and offers solutions for advanced and novice Python users. This web page contains a user manual and source code documentation.

pyfor is capable of processing large acquisitions of point data in just a few lines of code. Here is an example for performing a routine normalization for an entire collection of *.las* tiles.

```python
import pyfor
collection = pyfor.collection.from_dir('./my_tiles')

def normalize(las_path):
   tile = pyfor.cloud.Cloud(las_path)
   tile.normalize()
   tile.write('{}_normalized.las'.format(tile.name))

collection.par_apply(normalize, by_file=True)
```

The above example only scratches the surface. Please see the Installation and Getting Started pages to learn more.

CHAPTER 1

Introduction

Welcome to pyfor, a Python module intended for processing large aerial LiDAR (and phodar) acquisitions for the use of large-scale forest inventories. This document serves as a general introduction to the package, its philosophy, and a bit of its history. If you would like to jump straight into analysis, feel free to skip over this "soft" document.

1.1 About the Author

I am Bryce Frank, a PhD student at Oregon State University. I work for the Forest Measurements and Biometrics Lab, which is run under the guidance of Dr. Temesgen Hailemariam. Our work focuses on developing statistical methodology for the analysis of forest resources at multiple scales. Some of the lab members work on small-scale issues, like biomass and taper modeling. My work, along with others, is focused on producing reliable estimates of forest attributes for large scale forest assessment.

1.2 Package History

I originally began pyfor as a class project to explore the use of object oriented programming in GIS. At the time, I had been programming in Python for about two years, but still struggled with some concepts like Classes, object inheritance, and the like. pyfor was a way for me to finally learn some of those concepts and implement them in an analysis environment that was useful for me. Around the Spring of 2017, I released the package on GitHub. At the time, the package was in very rough condition, was very inefficient, and only did a few rudimentary tasks.

Around the Spring of 2018 I found a bit of time to rework the package from the ground up. I was deeply inspired by the lidR package, which I used extensively for a few months. I think lidR is a great tool, and pyfor is really just an alternative way of doing many of the same tasks. However, I prefer to work in Python for many reasons, and I also prefer to do my own scripting, so lidR fell by the wayside for me for those reasons. Rather than keep my scripts locked up somewhere, I modified the early version of pyfor with my newest attempts. I am also indebted to Bob McGaughey's FUSION, which paved the way in terms of these sorts of software, and is still my go-to software package for production work.

1.3 Philosophy

pyfor started as a means for me to learn OOP, and I think the framework is a very natural way to work with LiDAR data from an interactive standpoint. In this way, pyfor is a bit of a niche tool that is really designed more for research - being able to quickly change parameters on the fly and get quick visual feedback about the results is important for tree detection and other tasks. Because I am a bit selfish when I develop, and I am mainly a researcher at this point in my career, this will be the main objective for the package for the time being.

However, I completely understand the desire for performant processing. As the structure of pyfor beings to solidify, more of my time can be spent on diagnosing performance issues within the package and optimizing toward that end. I think Python, specifically scientific Python packages, will continue to be a solid platform for developing reasonably performant code, if done well. It is unlikely that pyfor will achieve speeds equivalent to raw C++ or FORTRAN code, but I do not think it will be orders of magnitude away if functions are developed in a way that leverages some of these faster packages. This also comes with the benefit of increased maintainability, clarity and usability - goals that I feel are often overlooked in the GIS world.

1.4 Acknowledgements

A special thank you to Drs. Ben Weinstein, Francisco Mauro-Gutiérrez and Temesgen Hailemariam for their continued support and advice as this project matures.

CHAPTER 2

Installation

miniconda or Anaconda is required for your system before beginning. pyfor depends on many packages that are otherwise tricky and difficult to install (especially gdal and its bindings), and conda provides a quick and easy way to manage many different Python environments on your system simultaneously.

The following bash commands will install this branch of pyfor. It requires installation of miniconda (see above). This will install all of the prerequisites in that environment, named pyfor_env. pyfor depends on a lot of heavy libraries, so expect construction of the environment to take a little time.

```
git clone https://github.com/brycefrank/pyfor.git
cd pyfor
conda env create -f environment.yml

# For Linux / macOS:
source activate pyfor_env

# For Windows:
activate pyfor_env

pip install .
```

Following these commands, pyfor should load in an activated Python shell:

```
import pyfor
```

If you see no errors, you are ready to process.

CHAPTER 3

Getting Started

This document describes a few basic operations, such as reading, writing, and basic point cloud manipulations for a single points dataset.

3.1 Reading a Point Cloud

Reading a point cloud means instantiating a *Cloud* object. The *Cloud* object is the integral part of a point cloud analysis in pyfor. Instantiating a *Cloud* is simple:

```python
import pyfor
tile = pyfor.cloud.Cloud("../pyfortest/data/test.las")
```

Once we have an instance of our Cloud object we can explore some information regarding the point cloud. We can print the Cloud object for a brief summary of the data within.

```python
print(tile)
```

```
File Path: ../data/test.las
File Size: 6082545
Number of Points: 217222
Minimum (x y z): [405000.01, 3276300.01, 36.29]
Maximum (x y z): [405199.99, 3276499.99, 61.12]
Las Version: 1.3
```

An important attribute of all *Cloud* objects is *.data*. This represents the raw data and header information of a *Cloud* object. It is managed by a separate, internal class called *LASData* in the case of .las files and *PLYData* in the case of .ply files. These classes manage some monotonous reading, writing and updating tasks for us, and are generally not necessary to interact with directly. Still, it is important to know they exist.

3.2 Filtering Raw Points

Sometimes it is interesting to view the raw points. For a *Cloud* object, these are stored in a pandas dataframe in the *.data.points* attribute:

```
tile.data.points.head()
```

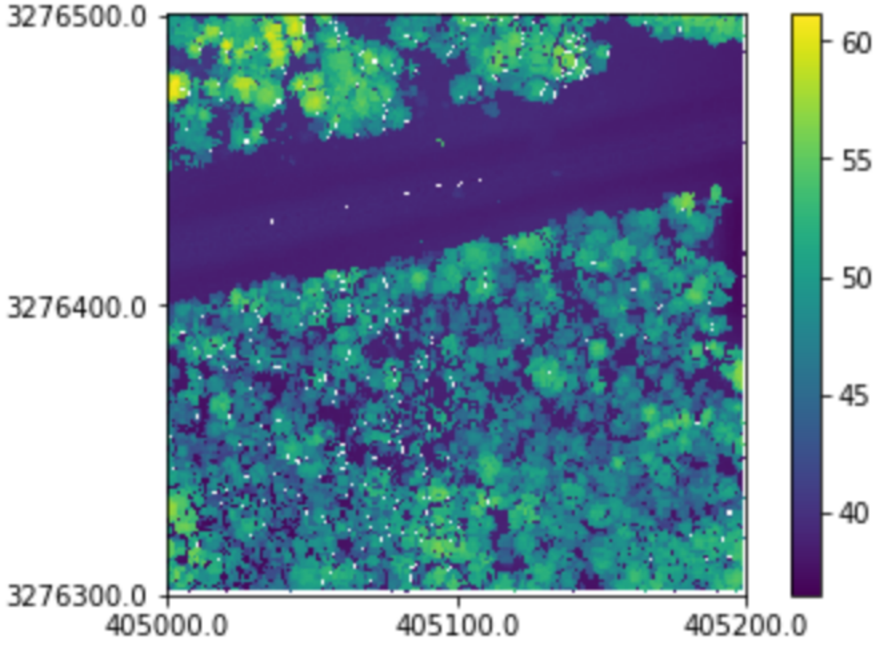
Direct modifications to the raw points should be done with caution, but is as simple as over-writing this dataframe. For example, to remove all points with an x dimension exceeding 405120:

```
tile.data.points = tile.data.points[tile.data.points["x"] < 405120]
```

3.3 Plotting

For quick visual inspection, a simple plotting method is available that uses *matplotlib* as a backend:

```
tile.plot()
```



3.4 Writing Points

Finally, we can write our point cloud out to a new file:

```
tile.write('my_new_tile.las')
```

CHAPTER 4

The Basics

4.1 Canopy Height Models

An integral part of any analysis is the production of a canopy height model, or CHM. The CHM is a rasterized representation of the canopy of the forest. The creation and filtering of CHMs play a large role in tree detection algorithms and are an interpretable way to display information.

A basic canopy height model can be created using a convenience wrapper:

```
tile = pyfor.cloud.Cloud('my_tile.las')
tile.normalize(1)
chm = tile.chm(0.5)
```

The above block will load the las file, *my_tile.las*, remove the ground elevation (normalize) and compute a basic canopy height model. Here, we specify a resolution of 0.5 units.

Note: In pyfor no assumptions are made about the reference system, so always specify resolutions in the units that the point cloud is registered in. In this case it was originally registered in meters, therefore the output raster will have a resolution of 0.5 meters.

4.1.1 Manipulating Canopy Height Models

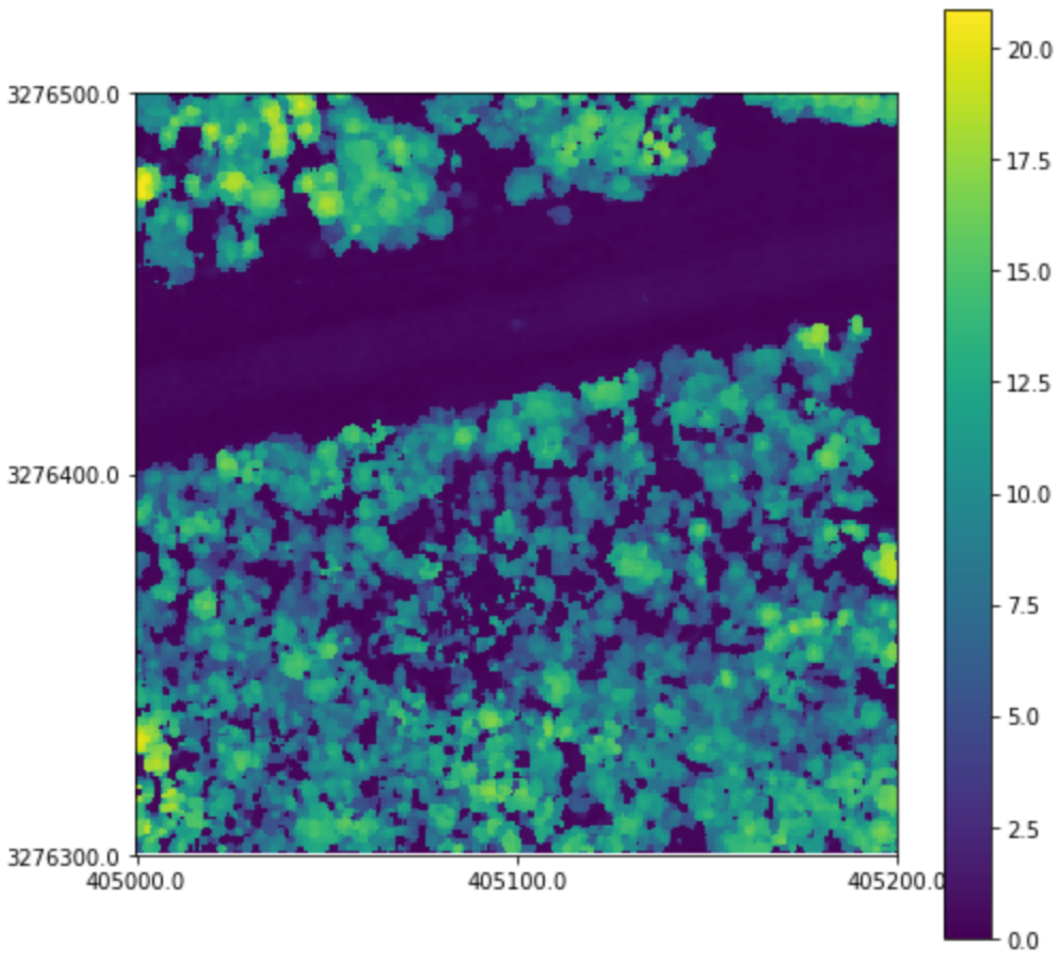
Often times, raw CHMs are not adequate for analysis. They contain many issues, such as missing values and data pits. We can add some extra arguments to add NaN interpolation and pit filtering.

```
better_chm = tile.chm(0.5, interp_method = "nearest", pit_filter = "median")
```

Here, we interpolate missing values using a nearest neighbor interpolator, and pass a median filter over the canopy height model to smooth pits.

We can display our CHM with the *.plot* method:

```
better_chm.plot()
```



4.1.2 Writing Canopy Height Models

A canopy height model is a *Raster* object. And can be written out in the same way.

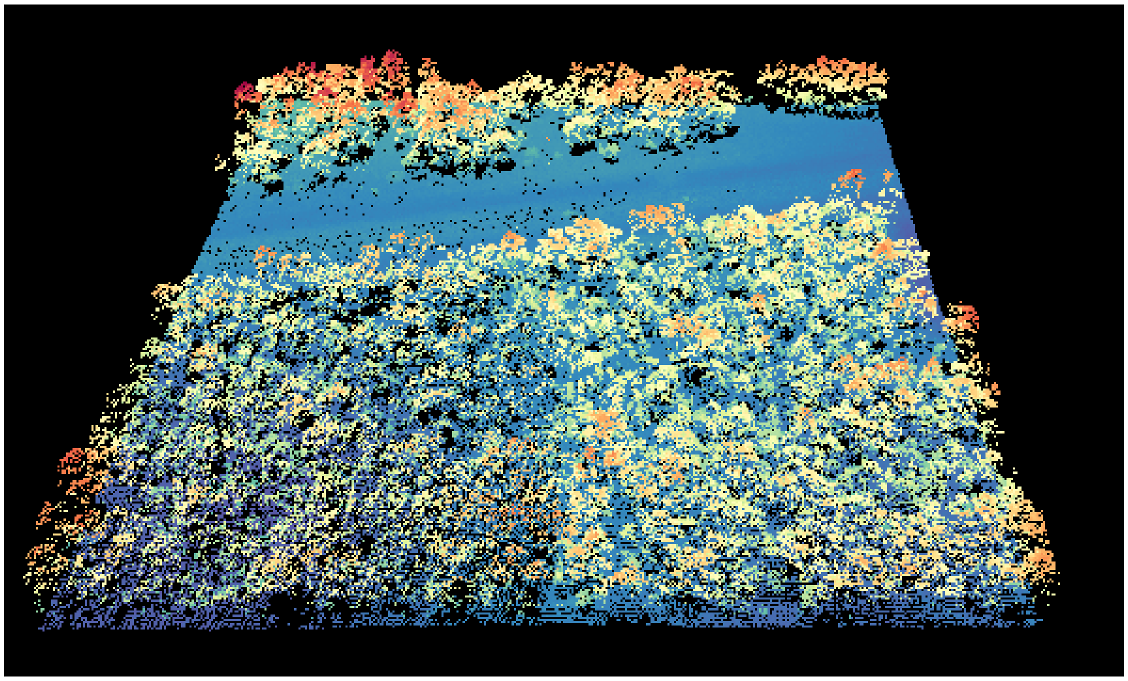
```
better_chm.write('my_chm.tif')
```

4.2 Clipping

Often times we want to clip out LiDAR points using a shapefile. This can be done using pyfor's Cloud.clip method. pyfor integrates with geopandas and shapely, convenient geospatial packages for Python, to provide a way to clip point clouds.

```
import pyfor
import geopandas

# Load point cloud
pc = pyfor.cloud.Cloud("../data/test.las")
pc.plot3d()
```

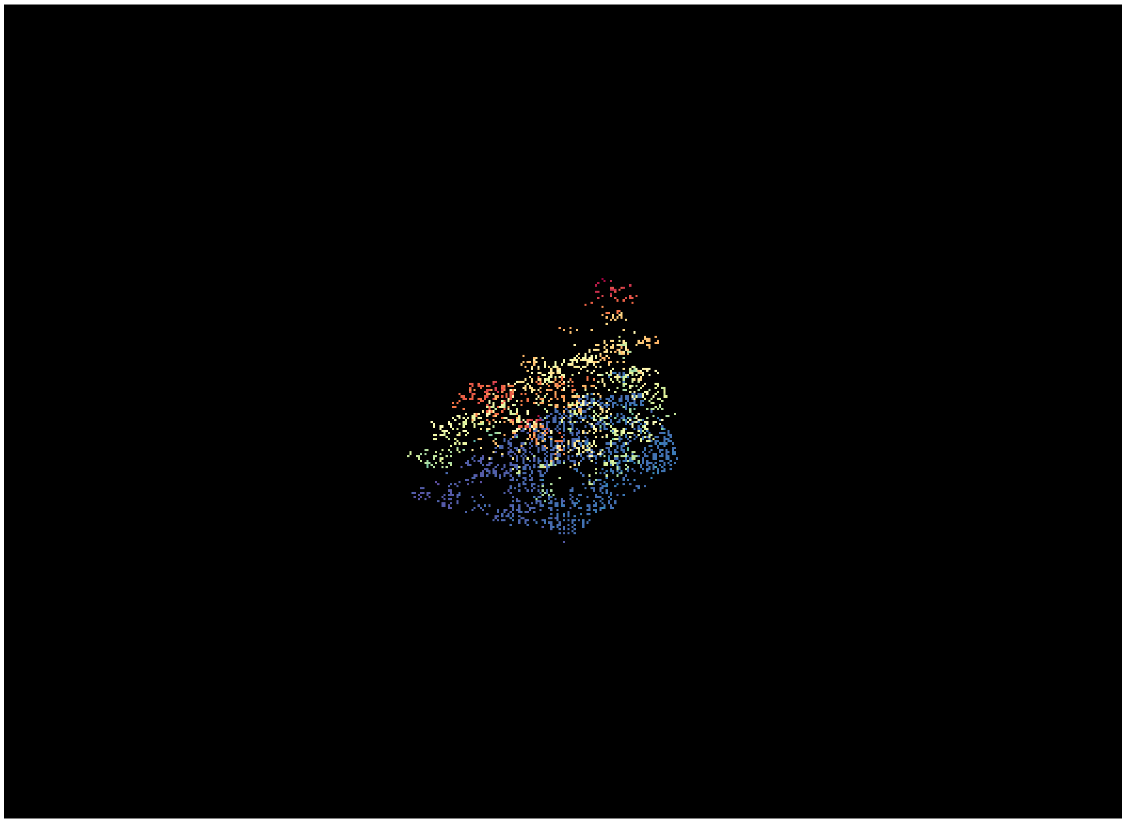


As input to the clipping function we need any *shapely.geometry.Polygon* our heart desires, as long as its coordinates correspond to the same physical space as the *Cloud* object. Here I extract a *Polygon* from a shapefile using *geopandas*:

```
# Load point cloud
polys = geopandas.read_file("../data/clip.shp")
poly = poly_frame["geometry"].iloc[0]
```

Finally, pass the *Polygon* to the clipping function. This function returns a new *Cloud* object.

```
# Load point cloud
clipped = pc.clip(poly)
clipped.plot3d()
```



4.3 Normalization

One of the most integral parts of LiDAR analysis is determining which points represent the ground. Once this step is completed, we can construct bare earth models (BEMs) and normalize our point clouds to produce reliable estimates of height, canopy height models and other products.

pyfor offers a few avenues for normalization, ground filtering and the creation of bare earth models. All of these methods are covered in the advanced Ground Filtering document. Here, only the convenience wrapper is covered.

The convenience wrapper *Cloud.normalize* is a function that filters the cloud object for ground points, creates a bare earth model, and uses this bare earth model to normalize the object in place. That is, **it conducts the entire normalization process from the raw data and does not leverage existing classified ground points**. See the advanced Ground Filtering document to use existing classified ground points.

It uses the *Zhang2003* filter by default and takes as its first argument the resolution of the bare earth model:

```
import pyfor
tile = pyfor.cloud.Cloud('my_tile.las')
tile.normalize(1)
```

4.4 Area-Based Metrics

The area-based approach (ABA) is ubiquitous in modern forest inventory systems. *pyfor* enables the computation of a set of area-based metrics for individual point clouds (*Cloud* objects) and for gridded point clouds (*Grid* objects).

The following block demonstrates a minimal example of creating standard metrics for a gridded tile.

```
import pyfor
tile = pyfor.cloud.Cloud('my_tile.las')
tile.normalize(1)
grid = tile.grid(20)
std_metrics = grid.standard_metrics(2)
```

This returns a Python dictionary, where each key is the name of the metric and each value is a *Raster* object. The argument *2* is the heightbreak at which canopy cover metrics are computed in meters.

```
{'max_z': <pyfor.rasterizer.Raster object at 0x000001BB2239C9B0>,
 'min_z': <pyfor.rasterizer.Raster object at 0x000001BB2239CD68>,
 ...
 'pct_all_above_mean': <pyfor.rasterizer.Raster object at 0x000001BB223E0EB8>}
```

Interacting with these key value pairs is natural, since the values are simply *Raster* objects. For example we can plot the *max_z* raster from the dictionary:

```
std_metrics['max_z'].plot()
```

topics/../img/max_z.png

Or, perhaps more useful, write the raster with a custom name:

```
std_metrics['max_z'].write('my_max_z.tif')
```

4.4.1 Standard Metrics Description

A number of metrics are included in the standard suite and are modeled heavily after FUSION software. Here is a brief description of each.

```
p_*: The height of the *th percentile along the z dimension.
     * = (1, 5, 10, 20, 25, 30, 40, 50, 60, 70, 75, 80, 90, 95, 99)
max_z: The maximum of the z dimension.
min_z: The minimum of the z dimension.
mean_z: The mean of the z dimension.
mean_z: The mean of the z dimension.
stddev_z: The standard deviation of the z dimension.
var_z: The variance of the z dimension.
canopy_relief_ratio: (mean_z - min_z) / (max_z - min_z)
pct_r_1_above_*: The percentage of first returns above a specified heightbreak.
pct_r_1_above_mean: The percentage of first returns above mean_z.
pct_all_above_*: The percentage of returns above a specified heightbreak.
pct_all_above_mean: The percentage of returns above mean_z.
```

CHAPTER 5

Advanced

5.1 Handling Large Acquisitions

The pyfor `collections` module allows for efficient handling of large acquisitions with paralell processing of user-define functions. This document presents a couple examples of large acquisition processing.

5.1.1 Creating Project-Level Canopy Height Models

A `CloudDataFrame` is the integral part of managing large point cloud acquisitions. It is initialized by passing a directory that contains `.las` or `.laz` files:

```
import pyfor
my_collection = pyfor.collection.from_dir("my_collection_dir", n_jobs=3)
```

Here we have instantiated a collection such that the processing function will be done in parallel across three cores.

It will be useful to set the coordinate reference system for the project. This is done via the `.crs` attribute:

```
import pyproj
crs = pyproj.proj({'init': 'epsg:26910'}).srs
my_collection.crs = crs
```

Here we must grapple with a few things. First, we will want to process buffered tiles to eliminate edge effects from normalization. This requires defining a few things for the collection.

First, we want to set the collection tiles such that the output rasters will be consistent for a specified grid cell size. This is done by manipulating the internal geometries stored in `my_collection.tiles`. By default, these describe the `.las/z` bounding boxes, but we want to ensure the output rasters are exactly the correct size so that they line up correctly. We can do this easily with the `.retile_raster` helper function. This is done **in place** for the collection.

```
my_collection.retile_raster(0.5, 500, buffer=20)
```

The first argument is the desired resolution of the output raster. The second argument is the resolution of the new tile sets. `500` means they will be processed in 500m x 500m chunks. Finally `buffer=20` means that each tile will be buffered by 20 meters, such that we can process a large buffered tile to eliminate edge effects.

Next, we want to define a function to process each buffered tile. This is where the flexibility of the collection enters in.

```
def my_process_func(buffered_cloud, tile):
    buffer_dist = 20
    buffered_cloud.normalize(1)

    # Generate CHM of buffered cloud
    chm = buffered_cloud.chm(0.5, interp_method="nearest", pit_filter="median")

    # Define output bounding box (remove the buffered part)
    coords = list(tile.exterior.coords)
    bbox = coords[0][0] + buffer_dist, coords[2][0] - buffer_dist, coords[0][1] +␣
↪buffer_dist, coords[1][
        1] - buffer_dist
    chm.force_extent(bbox)

    # Make a readable name for this particular tile
    flat_coords = [int(np.floor(coord)) for coord in bbox]
    tile_str = '{}_{}_{}_{}'.format(*flat_coords)

    # Write out the canopy height model
    chm.write('{}.tif'.format(tile_str))
```

The above function is lengthy, but really quite simple. First, we normalize and create a canopy height model for some given buffered point cloud. Then, we restrict its output to remove the buffer using `.force_extent`. Finally, we write out the canopy height model to file with a nicely formatted name.

Finally, we can execute the processing job with the following:

```
my_collection.par_apply(my_process_func)
```

CHAPTER 6

API Reference

6.1 pyfor package

6.1.1 Submodules

pyfor.clip module

pyfor.cloud module

pyfor.collection module

pyfor.gisexport module

pyfor.ground_filter module

pyfor.metrics module

pyfor.rasterizer module

pyfor.voxelizer module

6.1.2 Module contents

CHAPTER 7

Indices and Tables

- genindex
- modindex
- search