pyntcloud Documentation

Release 0.1.0

David de la Iglesia Castro

Oct 04, 2019

Contents

pyntcloud is a Python library for working with 3D point clouds.

This documentation is under construction.

CHAPTER 1

Introduction

This page will introduce the general concept of *point clouds* and illustrate the capabilities of pyntcloud as a point cloud processing tool.

1.1 Point clouds

Point clouds are one of the most relevant entities for representing three dimensional data these days, along with polygonal meshes (which are just a special case of point clouds with connectivity graph attached).

In its simplest form, a point cloud is a set of points in a cartesian coordinate system.

Accurate 3D point clouds can nowadays be (easily and cheaply) acquired from different sources. For example:

- RGB-D devices: Google Tango, Microsoft Kinect, etc.
- Lidar.
- Camera + Photogrammetry software (Open source Colmap, Agisoft Photoscan, . . .)

1.2 pyntcloud

pyntcloud enables simple and interactive exploration of point cloud data, regardless of which sensor was used to generate it or what the use case is.

Although it was built for being used on Jupyter Notebooks, the library is suitable for other kinds of uses.

pyntcloud is composed of several modules (as independent as possible) that englobe common point cloud processing operations:

- *Filters* / *Filters - Dev*
- geometry
- *I/O* / *I/O - Dev*

- learn
- neighbors
- plot
- ransac
- sampling
- *Scalar Fields* / *Scalar Fields - Dev*
- *Structures* / *Structures - Dev*
- utils

Most of the functionality of this modules can be accessed by the core class of the library, **PyntCloud**, and its corresponding methods:

```
from pyntcloud import PyntCloud
# io
cloud = PyntCloud.from_file("some_file.ply")
# structures
kdtree_id = cloud.add_structure("kdtree")
# neighbors
k_neighbors = cloud.get_neighbors(k=5, kdtree=kdtree_id)
# scalar_fields
ev = cloud.add_scalar_field("eigen_values", k_neighbors=k_neighbors)
# filters
f = cloud.get_filter("BBOX", min_x=0.1, max_x=0.8)
# ...
```

Although most of the functionality in the modules can be used without constructing a PyntCloud instance, the recommended workflow for the average user is the one showcased above.

CHAPTER 2

Installation

2.1 Minimum Requirements

At the very least, you need a Python(3) installation (an isolated environment, i.e conda, is recommended) and the following requirements:

- numpy
- numba
- scipy
- pandas

You can install this requirements however you want to although we recommend to use minconda:

https://conda.io/miniconda.html

And running:

```
conda create -n pyntcloud python=3 numpy numba scipy pandas

source activate pyntcloud
```

2.2 Basic Installation

Once you have this requirements installed, you can install pyntcloud using:

```
pip install git+https://github.com/daavoo/pyntcloud
```

2.3 Installation for developers

Check *Contributing*

CHAPTER 3

Contributing

If you want to hack around with PyntCloud you should install the depencies specified in *Installation*.

In addition to those, you need to install:

- flake8
- pytest

Then you can clone the repo and install it in editable mode:

```
git clone https://github.com/daavoo/pyntcloud.git
pip install -e pyntcloud
```

From the root of the repo, you can run:

```
# for getting warnings about syntax and other kinds of errors
flake8

# for running all the tests
pytest -v
```

CHAPTER 4

PyntCloud

PyntCloud is the core class that englobes almost all the functionality available in **pyntcloud**.

Whereas, in its classical form, the point clouds are understood as simple sets of points, a PyntCloud is a Python class with several **attributes** and **methods** that enable a more fluent way of manipulating this entity.

4.1 Attributes

PyntCloud's attributes serve to store information about the point cloud and the structures associated with it.

Each PyntCloud's instance some predefined attributes:

- centroid
- mesh
- *Points*
- structures
- xyz

You may also add other attributes to your own PyntCloud instance.

The information about this attributes is reported by the __repr__ method:

```
from pyntcloud import PyntCloud

cloud = PyntCloud.from_file("test/data/filters/filters.ply")

print(cloud)
```

```
PyntCloud
6 points with 0 scalar fields
0 faces in mesh
0 kdtrees
```

(continues on next page)

(continued from previous page)

```
0 voxelgrids
Centroid: 0.45000001788139343, 0.45000001788139343, 0.45000001788139343
Other attributes:
```

4.2 Methods

Available methods are very intuitive.

Methods starting with the word **add** incorporate new information to some existing PyntCloud attribute.

Methods starting with the word **get** return some information extracted from the PyntCloud.

I/O methods read/write information from/to different 3D formats.

Other methods are useful things that don't fit in any of the above categories.

4.2.1 *ADD* METHODS

`PyntCloud.`**`add_scalar_field`**`()`

Scalar Fields

Scalar Fields - Dev

`PyntCloud.`**`add_structure`**`()`

4.2.2 *GET* METHODS

`PyntCloud.`**`get_filter`**`()`

Filters

Filters - Dev

`PyntCloud.`**`get_sample`**`()`

Samplers

Samplers - Dev

`PyntCloud.`**`get_neighbors`**`()`

`PyntCloud.`**`get_mesh_vertices`**`()`

4.2.3 *I/O* METHODS

I/O

`PyntCloud.`**`from_file`**`()`

`PyntCloud.`**`to_file`**`()`

4.2.4 *OTHER* METHODS

`PyntCloud.`**`apply_filter`**`()`

`PyntCloud.`**`split_on`**`()`

`PyntCloud.`**`plot`**`()`

CHAPTER 5

Points

A classic point cloud is just a set of points.

In pyntcloud *points* is one of many attributes of the core class PyntCloud, although it's probably the most important.

This attribute is internally represented as a pandas DataFrame.

It is highly recommended to read the pandas DataFrame documentation in order to understand the possibilities for manipulating the point cloud information that this entity offers.

```
from pyntcloud import PyntCloud

cloud = PyntCloud.from_file("test/data/filters/filters.ply")

cloud.points
```

```
     x    y    z
0  0.0  0.0  0.0
1  0.1  0.1  0.1
2  0.2  0.2  0.2
3  0.5  0.5  0.5
4  0.9  0.9  0.9
5  1.0  1.0  1.0
```

5.1 Restrictions

Two of the few restrictions that you will find in pyntcloud are related to the *points* attribute.

- *points* must be a pandas DataFrame

If you want to instantiate a PyntCloud using the constructor, you have to pass a pandas DataFrame as the *points* argument.

If you want to change the *points* argument, you have to change it for a new pandas DataFrame.

```
import numpy as np

from pyntcloud import PyntCloud

points = np.random.rand(1000, 3)

cloud = PyntCloud(points)
```

```
TypeError: Points argument must be a DataFrame
```

- *points* must have 'x', 'y' and 'z' columns

The DataFrame that you use as *points* must have at least this 3 columns.

```
import numpy as np

import pandas as pd

from pyntcloud import PyntCloud

points = pd.DataFrame(np.random.rand(1000, 3))

cloud = PyntCloud(points)
```

```
ValueError: Points must have x, y and z coordinates
```

5.2 Basic manipulation

As mentioned above, the fact of having the points information in a pandas DataFrame brings many possibilities regarding the analysis and manipulation of this data.

As you can read in Working with scalar fields, one of the key features of pyntcloud is the flexibility that it offers regarding how you can add information and manipulate points.

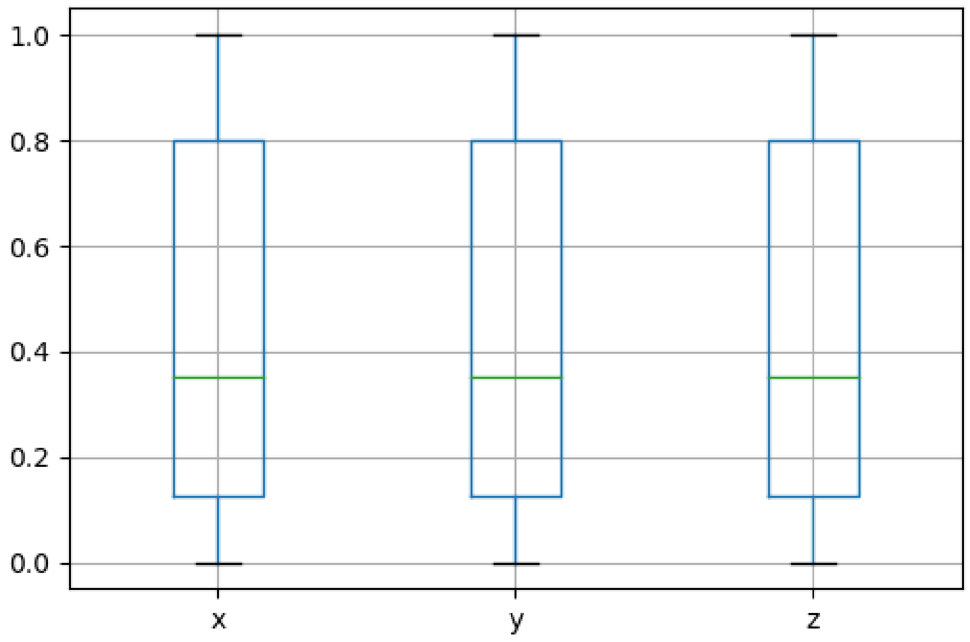
You can quickly get statistical information about points with a single command:

```
from pyntcloud import PyntCloud

cloud = PyntCloud.from_file("test/data/filters/filters.ply")

cloud.points.describe()
```

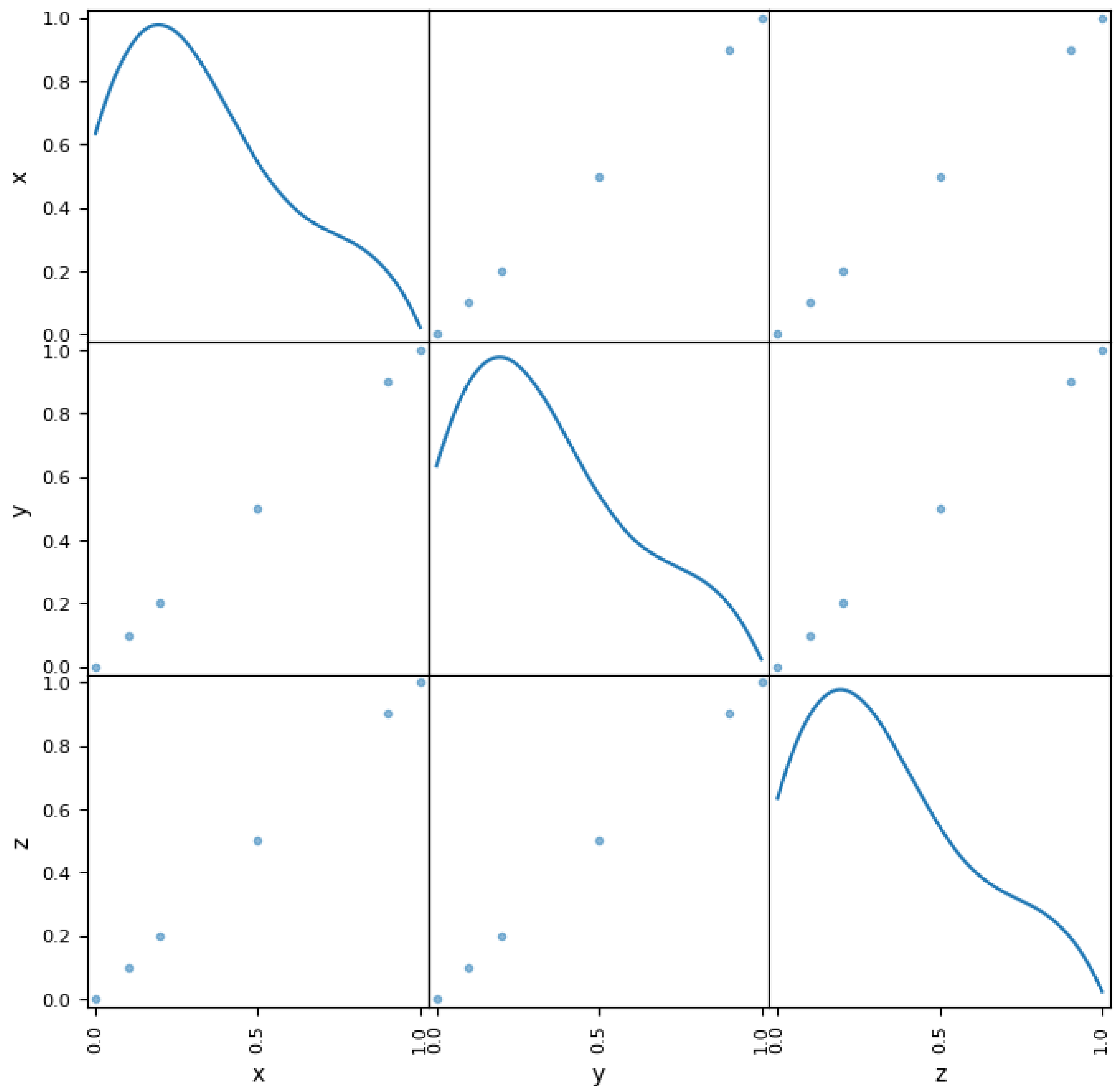
```
              x         y         z
count  6.000000  6.000000  6.000000
mean   0.450000  0.450000  0.450000
std    0.423084  0.423084  0.423084
min    0.000000  0.000000  0.000000
25%    0.125000  0.125000  0.125000
50%    0.350000  0.350000  0.350000
75%    0.800000  0.800000  0.800000
max    1.000000  1.000000  1.000000
```

You can use different plots to visualize the information in points:

```
cloud.points.boxplot()
```



```
from pandas.tools.plotting import scatter_matrix

scatter_matrix(cloud.points, diagonal="kde", figsize=(8,8))
```



You can perform operations over points like getting which points are above some "z" coordinate:

```
above_03 = cloud.points["z"] > 0.3
above_03
```

```
0     False
1     False
2     False
3      True
4      True
5      True
Name: x, dtype: bool
```

You can find more information about this kind of operation in Working with scalar fields.

As mentioned above, to fully understand the manipulation possibilities that the pandas DataFrame brings, is better to take a look at its documentation.

CHAPTER 6

Filters

As the name suggest, filters are used to discard points from the point cloud based on a condition that is evaluated against all the points in the point cloud.

All filters take a PyntCloud (and extra information in some cases) as input and produce a boolean array as output.

This boolean array separates the points that passed the filter and will thus be retained (True) from those which did not and will thus be removed (False).

Filters are accessible through:

`PyntCloud.get_filter()`

We group the available filters based on what the requirements for computing them are.

6.1 Only require XYZ

6.1.1 "BBOX"

6.2 Require KDTree

Required args:

kdtree: KDTree.id

```
kdtree = pointcloud.add_structure("kdtree", ...)
```

6.2.1 "ROR"

6.2.2 "SOR"

CHAPTER 7

Filters - Dev

This page contains useful information for developers who want to modify / add content to the filters module.

First of all, two points of advice:

- Use the existing filters as guideline.
- Follow the general CONTRIBUTING GUIDELINES.

7.1 The Big Picture

Filters are used by the method:

`PyntCloud.get_filter()`

Take a look at the source code in order to get a general overview of how filters are being used. All filters are classes and all have the same methods.

The sections below will guide you trough the filters module explaining how you can create your own filters or where you need to look in order to modify existing ones.

7.2 Base Class

All filters must inherit from the base class *Filter* and implement its abstract methods.

The base class is located at pyntcloud/filters/base.py

class `pyntcloud.filters.base.`**`Filter`**`(*, `*pyntcloud*`)`
  Base class for filters.

At the very least, all filters receive a PyntCloud when they are instantiated.

The *Filter.exctract_info* method must be overridden in order to extract and save the information required to compute the filter in a attribute.

See SUBMODULE BASE CLASS below for more information.

Filter.compute is where the boolean array is generated and returned. It should use the information extracted by the above method in order to decide which points should be filtered.

See SPECIFIC FILTER CLASS below.

7.3 Submodule Base Class

Filters are grouped into submodules according to which kind of information they require to be computed.

For example, filters that require a KDTree to be computed are in pyntcloud/filters/f_kdtree.py

As the information required by all the filters in each submodule is the same, we create a submodule base class overriding the __init__ and extract_info methods of the Filter base class.

For example, in the f_kdtree submodule there is a Filter_KDTree class from which all the filters that require a KDTree inherit.

If you don't find a submodule that extracts the information that your new filter needs, create a new one using as guideline one of the existing ones.

7.4 Specific Filter Class

Once you have a submodule base class that extracts the right information, you have to actually create the specific class for your filter, inheriting from the submodule base class and overriding the *Filter.compute* method.

If the computation of your filter requires some parameters from the user, you should override the *__init__* method in order to accept those parameters.

For example, the RadiusOutlierRemoval filter requires the user to specify a radius "r" and a number of neighbors "k":

7.5 Let PyntCloud know about your filter

In order to do so, you have to do some things:

- Add tests at *test/test_filters.py*.
- Import your new filter(s) and/or submodule(s) at *pyntcloud/filters/__init__.py*.
- Include them in the ALL_FILTERS dictionary, giving them a **string alias** at *pyntcloud/filters/__init__.py*.
- Document them in the *PyntCloud.get_filter* docstring at *pyntcloud/core_class.py*.
- Document them at *docs/filters.rst*.

CHAPTER 8

I/O

As mentioned in the introduction, 3D point clouds could be obtained from many different sources, each one with its own file format.

In addition to file formats used by each manufacturer, point clouds may also be stored in generic binary and ascii formats using different programming languages.

PyntCloud provides reading and writing routines for many common 3D file and generic array formats (more formats will be added in the near future):

- .asc / .pts / .txt / .csv / .xyz (see 'Note about ASCII files' below)
- .las
- .npy / .npz
- .obj
- .off (with color support)
- .pcd
- .ply

8.1 Reading

classmethod PyntCloud.**from_file**(*filename*, ***kwargs*)
Extract data from file and construct a PyntCloud with it.

Parameters

- **filename** (*str*) – Path to the file from which the data will be read
- **kwargs** (*only usable in some formats*) –

Returns **PyntCloud** – PyntCloud instance, containing all valid elements in the file.

Return type object

```
from pyntcloud import PyntCloud
my_point_cloud = PyntCloud.from_file("some_file.ply")
```

8.2 Writing

`PyntCloud.to_file`(*filename*, *also_save=None*, ***kwargs*)

Save PyntCloud data to file.

Parameters

- **filename** (*str*) – Path to the file from which the data will be read
- **also_save** (*list of str, optional*) – Default: None Names of the attributes that will be extracted from the PyntCloud to be saved in addition to points. Usually also_save=[“mesh”]
- **kwargs** (*only usable in some formats*) –

```
# my_point_cloud is a PyntCloud instance
my_point_cloud.to_file("out_file.obj", internal=["points", "mesh"])
```

8.3 Alternative ways for creating PyntClouds

Even though PyntCloud includes readers for some of the most common 3D file formats, there are many other formats and sources that you can use to store point cloud data.

That’s why although PyntCloud will include support for other file formats, it will never cover all.

The good news is that as long as you can **load the data into Python**, you can create a PyntCloud instance manually.

The key thing to understand is that you can’t just plug in the raw data into the PyntCloud constructor; there are some restrictions.

These restrictions are covered in *Points*.

As long as you can adapt your data to these restrictions, you will be able to construct a PyntCloud from formats that are not covered in *from_file*.

8.4 Note about ASCII files

There are many formats englobed in these kinds of files: .asc, .txt, .pts, . . .

Normally, the first 3 columns represent the X,Y,Z coordinates of the point and the rest of the columns represent some scalar field associated to that point (Maybe R,G,B values, or Nx,Ny,Nz, etc). But there is no official format specification.

Given all the possibilities that this brings, *PyntCloud.from_file* accepts keyword arguments in order to let the user adjust the loading for every possibility.

Internally, PyntCloud.from_file is just calling the pandas function .read_csv when you give it a valid ascii format.

So check the linked documentation to explore all the possible arguments in order to adjust them to read your ascii file.

For example, given a *example.pts* file with this content:

```
8
-0.037829 0.12794 0.004474
-0.044779 0.128887 0.001904
-0.068009 0.151244 0.037195
-0.002287 0.13015 0.02322
-0.022605 0.126675 0.007155
-0.025107 0.125921 0.006242
-0.03712 0.127449 0.001795
0.033213 0.112692 0.027686
```

You can construct a PyntCloud as follows:

```
import pandas as pd
from pyntcloud import PyntCloud

cloud = PyntCloud.from_file("example.pts",
                            sep=" ",
                            header=0,
                            names=["x","y","z"])
```

CHAPTER 9

I/O - Dev

CHAPTER 10

Samplers

Samplers use PyntCloud information to generate a sample of points. These points might or might not have been present in the original point cloud.

For example, *RandomPoints* generates a sample by randomly selecting points from the original point cloud. In this case all sample's points were present in the original point cloud.

On the other hand, *VoxelgridCentroids* generates a sample by computing the centroid of each group of points inside of each occupied VoxelGrid's cell. In this case any of the sample's points were present in the original point cloud.

All samplers take a point cloud as input and return a pandas DataFrame.

This pandas DataFrame can be used to generate a new PyntCloud.

All samplers are accessible trough:

`PyntCloud.get_sample()`

We group the available samplers based on what information is used for their computation.

10.1 Require points

10.1.1 "points_random_sampling"

10.2 Require mesh

pointcloud.mesh must exists.

10.2.1 “mesh_random_sampling”

10.3 Require VoxelGrid

Required args:

voxelgrid: VoxelGrid.id

```
voxelgrid = pointcloud.add_structure("voxelgrid", ...)
```

10.3.1 “voxelgrid_centers”

10.3.2 “voxelgrid_centroids”

10.3.3 “voxelgrid_nearest”

CHAPTER 11

Samplers - Dev

This page contains useful information for developers who want to modify / add content to the samplers module.

First of all, two points of advice:

- Use the existing samplers as guideline.
- Follow the general CONTRIBUTING GUIDELINES.

11.1 The Big Picture

Filters are used by the method:

`PyntCloud.get_sampler()`

Take a look at the source code in order to get a general overview of how samplers are being used. All samplers are classes and all have the same methods.

The sections below will guide you trough the samplers module explaining how you can create your own samplers or where you need to look in order to modify existing ones.

11.2 Base Class

All samplers must inherit from the base class *Sampler* and implement its abstract methods.

The base class is located at pyntcloud/filters/base.py

class `pyntcloud.samplers.base.`**`Sampler`**`(*, `*pyntcloud*`)`
  Base class for sampling methods.

At the very least, all samplers receive a PyntCloud when they are instantiated.

The *Sampler.exctract_info* method must be overridden in order to extract and save the information required to generate the sample in a attribute.

See SUBMODULE BASE CLASS below for more information.

Sampler.compute is where the sample is generated and returned. It should use the information extracted by the above method in order to generate the sample.

See SPECIFIC SAMPLER CLASS below.

11.3 Submodule Base Class

Samplers are grouped into submodules according to which kind of information they require to be computed.

For example, samplers that require a VoxelGrid to be computed are in pyntcloud/samplers/s_voxelgrid.py

As the information required by all the filters in each submodule is the same, we create a submodule base class overriding the __init__ and exctract_info methods of the Sampler base class.

For example, in the s_voxelgrid submodule there is a Sampler_Voxelgrid class from which all the samplers that require a Voxelgrid inherit.

If you don't find a submodule that extracts the information that your new sampler needs, create a new one using as guideline one of the existing ones.

11.4 Specific Sampler Class

Once you have a submodule base class that extracts the right information, you have to actually create the specific class for your sampler, inheriting from the submodule base class and overriding the *Sampler.compute* method.

If the computation of your sample requires some parameters from the user, you should override the *__init__* method in order to accept those parameters.

For example, the *RandomMesh* sampler requires the user to specify if the sample will use RGB and/or normal information:

11.5 Let PyntCloud know about your sampler

In order to do so, you have to do some things:

- Add tests at *test/test_samplers.py*.
- Import your new sampler(s) and/or submodule(s) at *pyntcloud/samplers/__init__.py*.
- Include them in the ALL_SAMPLERS dictionary, giving them a **string alias**, at *pyntcloud/samplers/__init__.py*.
- Document them in the *PyntCloud.get_sample* docstring at *pyntcloud/core_class.py*.
- Document them at *docs/samplers.rst*.

CHAPTER 12

Structures

Structures are used for adding superpowers to PyntCloud intances.

For example, a *VoxelGrid* can be used for:

- Converting a point cloud into a valid input for a convolutional neural network.
- Finding nearest neighbors.
- Finding unconnected clusters of points in the point cloud.
- Many other cool things.

All structures are built on top of a point cloud, mesh or another structure.

You can create structures using:

`PyntCloud.add_structure()`

12.1 Convex Hull

class `pyntcloud.structures.`**`ConvexHull`**`(`*points*, *incremental=False*, *qhull_options=None*`)`

12.2 Delaunay3D

class `pyntcloud.structures.`**`Delaunay3D`**`(`*points*, *furthest_site=False*, *incremental=False*, *qhull_options=None*`)`

12.3 KDTree

class `pyntcloud.structures.`**`KDTree`**`(`**, points*, *leafsize=16*, *compact_nodes=False*, *balanced_tree=False*`)`

12.4 VoxelGrid

class pyntcloud.structures.**VoxelGrid**(**, points, n_x=1, n_y=1, n_z=1, size_x=None, size_y=None, size_z=None, regular_bounding_box=True*)

CHAPTER 13

Structures - Dev

CHAPTER 14

Scalar Fields

Roughly speaking, each of the columns in the *PyntCloud.points* DataFrame is a Scalar Field.

Point clouds require at least 3 columns to be defined (the x,y,z coordinates); any other information associated to each point is a Scalar Field.

For example point clouds with color information usually have 3 scalar fields representing the Red, Green, and Blue values for each point.

A Scalar Field must have the same type and meaning for every point, the value is what is variable.

In the point cloud literature, Scalar Fields are restricted to be numeric (that's where the Scalar comes from), but here we extend the Scalar Field term to defined any column of the Points DataFrame.

Scalar Fields are accessible trough:

`PyntCloud.`**`add_scalar_field`**`()`

We group the available scalar fields based on what the requirements for computing them are.

14.1 Only require XYZ

14.1.1 "plane_fit"

`class` `pyntcloud.scalar_fields.`**`PlaneFit`**`(`*, *pyntcloud*, *max_dist=0.0001*, *max_iterations=100*, *n_inliers_to_stop=None*`)`

Get inliers of the best RansacPlane found.

14.1.2 "sphere_fit"

`class` `pyntcloud.scalar_fields.`**`SphereFit`**`(`*, *pyntcloud*, *max_dist=0.0001*, *max_iterations=100*, *n_inliers_to_stop=None*`)`

Get inliers of the best RansacSphere found.

14.1.3 "custom_fit"

class pyntcloud.scalar_fields.**CustomFit**(*pyntcloud, model, sampler, name, model_kwargs={}, sampler_kwargs={}, max_iterations=100, n_inliers_to_stop=None*)

Get inliers of the best custom model found.

14.1.4 "spherical_coords"

class pyntcloud.scalar_fields.**SphericalCoordinates**(**, pyntcloud, degrees=True*)

Get radial, azimuthal and polar values.

14.2 Require Eigen Values

Required args:

ev: list of str

```
ev = pointcloud.add_scalar_field("eigen_values", ...)
```

14.2.1 "anisotropy"

class pyntcloud.scalar_fields.**Anisotropy**(**, pyntcloud, ev*)

14.2.2 "curvature"

class pyntcloud.scalar_fields.**Curvature**(**, pyntcloud, ev*)

14.2.3 "eigenentropy"

class pyntcloud.scalar_fields.**Eigenentropy**(**, pyntcloud, ev*)

14.2.4 "eigen_sum"

class pyntcloud.scalar_fields.**EigenSum**(**, pyntcloud, ev*)

14.2.5 "linearity"

class pyntcloud.scalar_fields.**Linearity**(**, pyntcloud, ev*)

14.2.6 "ommnivariance"

class pyntcloud.scalar_fields.**Omnivariance**(**, pyntcloud, ev*)

14.2.7 “planarity”

class pyntcloud.scalar_fields.**Planarity**(**, pyntcloud, ev*)

14.2.8 “sphericity”

class pyntcloud.scalar_fields.**Planarity**(**, pyntcloud, ev*)

14.3 Require K Neighbors

Required args:

k_neighbors: (N, k) ndarray

```
k_neighbros = pointcloud.get_k_neighbors(k=10, ...)
```

14.3.1 “normals”

14.3.2 “eigen_values”

class pyntcloud.scalar_fields.**EigenValues**(**, pyntcloud, k_neighbors*)
Compute the eigen values of each point’s neighbourhood.

14.3.3 “eigen_decomposition”

class pyntcloud.scalar_fields.**EigenDecomposition**(**, pyntcloud, k_neighbors*)
Compute the eigen decomposition of each point’s neighbourhood.

14.4 Require Normals

pointcloud.points must have [nx, ny, nz] columns.

14.4.1 “inclination_deg”

class pyntcloud.scalar_fields.**InclinationDegrees**(**, pyntcloud*)
Vertical inclination with respect to Z axis in degrees.

14.4.2 “inclination_rad”

class pyntcloud.scalar_fields.**InclinationRadians**(**, pyntcloud*)
Vertical inclination with respect to Z axis in radians.

14.4.3 “orientation_deg”

class pyntcloud.scalar_fields.**OrientationDegrees**(**, pyntcloud*)
Horizontal orientation with respect to the XY plane in degrees.

14.4.4 "orientation_rad"

class pyntcloud.scalar_fields.**OrientationRadians**(*, *pyntcloud*)
Horizontal orientation with respect to the XY plane in radians.

14.5 Require RGB

pointcloud.points must have [red, green, blue] columns.

14.5.1 "hsv"

class pyntcloud.scalar_fields.**HueSaturationValue**(*, *pyntcloud*)
Hue, Saturation, Value colorspace.

14.5.2 "relative_luminance"

class pyntcloud.scalar_fields.**RelativeLuminance**(*, *pyntcloud*)
Similar to grayscale. Computed following Wikipedia.

14.5.3 "rgb_intensity"

class pyntcloud.scalar_fields.**RGBIntensity**(*, *pyntcloud*)
Red, green and blue intensity.

14.6 Require VoxelGrid

Required args:

voxelgrid: VoxelGrid.id

```
voxelgrid = self.add_structure("voxelgrid", ...)
```

14.6.1 "euclidean_clusters"

class pyntcloud.scalar_fields.**EuclideanClusters**(*, *pyntcloud*, *voxelgrid_id*)
Assing corresponding cluster to each point inside each voxel.

14.6.2 "voxel_n"

class pyntcloud.scalar_fields.**VoxelN**(*, *pyntcloud*, *voxelgrid_id*)
Voxel index in 3D array using 'C' order.

14.6.3 "voxel_x"

class pyntcloud.scalar_fields.**VoxelX**(*, *pyntcloud*, *voxelgrid_id*)
Voxel index along x axis.

14.6.4 "voxel_y"

class pyntcloud.scalar_fields.**VoxelY**(*, *pyntcloud*, *voxelgrid_id*)

Voxel index along y axis.

14.6.5 "voxel_z"

class pyntcloud.scalar_fields.**VoxelZ**(*, *pyntcloud*, *voxelgrid_id*)

Voxel index along z axis.

CHAPTER 15

Scalar Fields - Dev

This page contains useful information for developers that want to modify / add content to the scalar_fields module.

First of all, two points of advice:

- Use the existing scalar_fields as guideline.
- Follow the general CONTRIBUTING GUIDELINES.

15.1 The Big Picture

Filters are used by the method:

`PyntCloud.`**`add_scalar_field`**`()`

Take a look at the source code in order to get a general overview of how scalar fields are being used. All scalar fields are classes and all have the same methods.

The sections below will guide you trough the scalar fields module explaining how you can create your own scalar fields or where you need to look in order to modify existing ones.

15.2 Base Class

All filters must inherit from the base class *ScalarField* and implement its abstract methods.

The base class is located at pyntcloud/scalar_fields/base.py

`class` `pyntcloud.scalar_fields.base.`**`ScalarField`**`(*, `*`pyntcloud`*`)`
    Base class for scalar fields.

At the very least, all scalar fields receive a PyntCloud when they are instantiated.

The *ScalarField.exctract_info* method must be overridden in order to extract and save in a attribute the information required to compute the scalar field.

See SUBMODULE BASE CLASS below for more information.

ScalarField.compute is where the new DataFrame columns are generated.

See SPECIFIC SCALAR FIELD CLASS below.

15.3 Submodule Base Class

Scalar fields are grouped in submodules according to which kind of information they require to be computed.

For example, scalar fields that require a VoxelGrid to be computed are in pyntcloud/scalar_fields/sf_voxelgrid.py

As the information required by all the scalar fields in each submodule is the same, we create a submodule base class overriding the __init__ and exctract_info methods of the ScalarField base class.

For example, in the sf_voxelgrid submodule there is a ScalarField_Voxelgrid class from which all the scalar fields that require a VoxelGrid inherit.

If you don't find a submodule that extracts the information that your new scalar field needs, create a new one using as guideline one of the existing ones.

15.4 Specific Scalar Field Class

Once you have a submodule base class that extracts the right information, you have to actually create the specific class for your scalar field, inheriting from the submodule base class and overriding the *ScalarField.compute* method.

If the computation of your scalar requires some parameters from the user, you should override the *__init__* method in order to accept those parameters.

For example, the SphericalCoordinates scalar field (in pyntcloud/scalar_fields/sf_xyz.py) requires the user to specify if the output should be in degrees or not:

15.5 Let PyntCloud know about your filter

In order to do so, you have to do some things:

- Add tests at *test/test_sf.py*.
- Import your new scalar field(s) and/or submodule(s) at *pyntcloud/scalar_fields/__init__.py*.
- Include them in the ALL_SF dictionary, giving them a **string alias** at *pyntcloud/scalar_fields/__init__.py*.
- Document them in the *PyntCloud.add_scalar_field* docstring at *pyntcloud/core_class.py*.
- Document them at *docs/scalar_fields.rst*.

Index