py4dgeo

Release 0.6.0

Dominic Kempf

May 14, 2024

CONTENTS:

py4dgeo is a C++ library with Python bindings for change analysis in multitemporal and 4D point clouds.

Topographic 3D/4D point clouds are omnipresent in geosciences, environmental, ecological and archaeological sciences, robotics, and many more fields and applications. Technology to capture such data using laser scanning and photogrammetric techniques have evolved into standard tools. Dense time series of topographic point clouds are becoming increasing available and require tools for automatic analysis. Moreover, methods considering the full 4D (3D space + time) data are being developed in research and need to be made available in an accessible way with flexible integration into existent workflows.

The **main objective** of py4dgeo is to bundle and provide different methods of 3D/4D change analysis in a dedicated, comprehensive Python library. py4dgeo is designed as an international open source project that can be integrated into almost any 3D and GIS software in the geodata domain supporting Python, e.g. as plugins.

py4dgeo is under *ongoing active development*. Below, you find a list of provided methods.

CHAPTER

ONE

METHODS PROVIDED BY PY4DGEO

- **M3C2 algorithm** (Lague et al., 2013) for bitemporal point cloud distance computation. The concept and algorithm is explained in this tutorial by James Dietrich, including usage in the graphical software CloudCompare.
- **M3C2-EP** (M3C2-EP; Winiwarter et al., 2021) for statistical signal-noise separation in change analysis through error propagation. The concept and method are explained in full detail in the related paper.
- **4D objects-by-change** (4D-OBC; Anders et al., 2021) for time series-based extraction of surface activities *[under active development]*. The concept and method are explained in this scientific talk:
- **Correspondence-driven plane-based M3C2** (Zahs et al., 2022) for lower uncertainty in 3D topographic change quantification. The concept and method are explained in this scientific talk:
- **Point cloud registration**: Py4dgeo supports to calculate and apply affine transformations to point clouds using a standard ICP implementations. More ICP methods are currently being implemented - stay tuned!

CHAPTER

TWO

EXAMPLES

2.1 Demo notebooks using methods provided by py4dgeo

CHAPTER

THREE

INSTALLATION

3.1 Prerequisites

Using py4dgeo requires the following software installed:

- 64-bit Python `>= 3.8` (32-bit installations might cause trouble during installation of dependencies)

In order to build the package from source, the following tools are also needed.

- A C++17-compliant compiler
- CMake `>= 3.9`
- Doxygen (optional, documentation building is skipped if missing)

3.2 Installing py4dgeo

The preferred way of installing `py4dgeo` is using `pip`.

3.3 Installing the release version using pip

`py4dgeo` can be installed using `pip` to obtain the current release:

```
python -m pip install py4dgeo
```

3.4 Building from source using pip

The following sequence of commands is used to build `py4dgeo` from source:

```
git clone --recursive https://github.com/3dgeo-heidelberg/py4dgeo.git
cd py4dgeo
python -m pip install -v --editable .
```

The `--editable` flag allows you to change the Python sources of `py4dgeo` without reinstalling the package. The `-v` flag enables verbose output which gives you detailed information about the compilation process that you should include into potential bug reports. To recompile the C++ source, please run `pip install` again. In order to enable multi-threading on builds from source, your compiler toolchain needs to support `OpenMP`.

If you want to contribute to the library's development you should also install its additional Python dependencies for testing and documentation building:

```
python -m pip install -r requirements-dev.txt
```

3.5 Setting up py4dgeo using Docker

Additionally, `py4dgeo` provides a Docker image that allows to explore the library using JupyterLab. The image can be locally built and run with the following commands:

```
docker build -t py4dgeo:latest .
docker run -t -p 8888:8888 py4dgeo:latest
```

CHAPTER

FOUR

DOCUMENTATION OF SOFTWARE USAGE

As a starting point, please have a look to the Jupyter Notebooks available in the repository and find the `py4dgeo` documentation on readthedocs.

CHAPTER

FIVE

PUBLISHED TEST DATA

If you are looking for data to test different methods, consider the following open data publications:

5.1 Hourly TLS point clouds of a sandy beach

 Vos et al. (2022): [https://doi.org/10.1038/s41597-022-01291-9{]}(https://doi.org/10.1038/s41597-022-01291-9).

5.2 By-weekly TLS point clouds of an Alpine rock glacier

 Zahs et al. (2022): [https://doi.org/10.11588/data/TGSVUI{]}(https://doi.org/10.11588/data/TGSVUI).

CHAPTER

SIX

CITATION

Please cite py4dgeo when using it in your research and reference the appropriate release version.

```
article{py4dgeo,
author = {py4dgeo Development Core Team}
title = {py4dgeo: library for change analysis in 4D point clouds},
journal = {},
year = {2022},
number = {},
volume = {},
doi = {},
url = {https://github.com/3dgeo-heidelberg/py4dgeo},
}
```

CHAPTER

SEVEN

FUNDING / ACKNOWLEDGEMENTS

The initial software development was supported by the **Scientific Software Center (SSC)** in the Open Call 2021. The scientific software project is further supported by the research projects **CharAct4D** and **AImon5.0**.

CHAPTER

EIGHT

CONTACT / BUGS / FEATURE REQUESTS

You think you have found a bug or have specific request for a new feature? Please open a new issue in the online code repository on Github. Also for general questions please use the issue system.

Scientific requests can be directed to the 3DGeo Research Group Heidelberg and its respective members.

CHAPTER

NINE

LICENSE

See LICENSE.md.

CHAPTER

TEN

LITERATURE

- Anders, K., Winiwarter, L., Mara, H., Lindenbergh, R., Vos, S.E. & Höfle, B. (2021): Fully automatic spatiotemporal segmentation of 3D LiDAR time series for the extraction of natural surface changes. ISPRS Journal of Photogrammetry and Remote Sensing, 173, pp. 297-308. DOI: 10.1016/j.isprsjprs.2021.01.015.
- Lague, D., Brodu, N., & Leroux, J. (2013). Accurate 3D comparison of complex topography with terrestrial laser scanner: Application to the Rangitikei canyon (N-Z). ISPRS Journal of Photogrammetry and Remote Sensing, 82, pp. 10-26. DOI: 10.1016/j.isprsjprs.2013.04.009.
- Winiwarter, L., Anders, K., Höfle, B. (2021): M3C2-EP: Pushing the limits of 3D topographic point cloud change detection by error propagation. ISPRS Journal of Photogrammetry and Remote Sensing, 178, pp. 240–258. DOI: 10.1016/j.isprsjprs.2021.06.011.
- Zahs, V., Winiwarter, L., Anders, K., Williams, J.G., Rutzinger, M. & Höfle, B. (2022): Correspondence-driven plane-based M3C2 for lower uncertainty in 3D topographic change quantification. ISPRS Journal of Photogrammetry and Remote Sensing, 183, pp. 541-559. DOI: 10.1016/j.isprsjprs.2021.11.018.

CHAPTER

ELEVEN

BASIC USAGE TUTORIALS

In the following, you find a number of tutorials that demonstrate the basic capabilities of `py4dgeo`.

11.1 Basic M3C2 algorithm

This presents how the M3C2 algorithm (*Lague et al., 2013*) for point cloud distance computation can be run using the `py4dgeo` package. As a first step, we import the `py4dgeo` and `numpy` packages:

[1]:
```
import numpy as np
import py4dgeo
```

Next, we need to load two datasets that cover the same scene at two different points in time. Point cloud datasets are represented by `numpy` arrays of shape `n x 3` using a 64 bit floating point type (`np.float64`). Here, we work with a rather small synthetical data set:

[2]:
```
epoch1, epoch2 = py4dgeo.read_from_xyz(
    "plane_horizontal_t1.xyz", "plane_horizontal_t2.xyz"
)
```

```
[2024-05-14 13:09:45][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t1.xyz'
[2024-05-14 13:09:45][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t2.xyz'
```

The analysis of point cloud distances is executed on so-called *core points* (cf. Lague et al., 2013). These could be, e.g., one of the input point clouds, a subsampled version thereof, points in an equidistant grid, etc. Here, we choose a subsampling by taking every 50th point of the reference point cloud:

[3]:
```
corepoints = epoch1.cloud[::50]
```

Next, we instantiate the algorithm class and run the distance calculation:

[4]:
```
m3c2 = py4dgeo.M3C2(
    epochs=(epoch1, epoch2),
    corepoints=corepoints,
    cyl_radii=(2.0,),
    normal_radii=(0.5, 1.0, 2.0),
)

distances, uncertainties = m3c2.run()
```

```
[2024-05-14 13:09:45][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:09:45][INFO] Building KDTree structure with leaf parameter 10
```

The calculated result is an array with one distance per core point. The order of distances corresponds exactly to the order of input core points.

[5]:
```
distances
```

[5]:
```
array([-0.10269298, -0.09957772, -0.10179986, -0.10063675, -0.10091512,
       -0.10070981, -0.09819643, -0.09926434, -0.09911222])
```

Corresponding to the derived distances, an uncertainty array is returned which contains several quantities that can be accessed individually: The level of detection `lodetection`, the spread of the distance across points in either cloud (`spread1` and `spread2`, by default measured as the standard deviation of distances) and the total number of points taken into consideration in either cloud (`num_samples1` and `num_samples2`):

[6]:
```
uncertainties["lodetection"]
```

[6]:
```
array([0.00565384, 0.00181819, 0.00255157, 0.00332511, 0.00281475,
       0.00299644, 0.00295429, 0.00199006, 0.00360769])
```

[7]:
```
uncertainties["spread1"]
```

[7]:
```
array([0.00417673, 0.00193209, 0.00203436, 0.00368965, 0.00259167,
       0.00286324, 0.0032822 , 0.00247362, 0.00356988])
```

[8]:
```
uncertainties["num_samples1"]
```

[8]:
```
array([ 5, 11, 11, 11, 12, 10, 11, 12, 10])
```

11.1.1 References

- Lague, D., Brodu, N., & Leroux, J. (2013). Accurate 3D comparison of complex topography with terrestrial laser scanner: Application to the Rangitikei canyon (N-Z). ISPRS Journal of Photogrammetry and Remote Sensing, 82, pp. 10-26. doi: 10.1016/j.isprsjprs.2013.04.009.

11.2 Algorithm customization

`py4dgeo` does not only provide a high performance implementation of the M3C2 base algorithm and some of it variants. It also allows you to rapidly prototype new algorithms in Python. We will demonstrate the necessary concepts by implementing some dummy algorithms without geographic relevance. First, we do the necessary setup, please consult the *M3C2 notebook* for details.

[1]:
```
import numpy as np
import py4dgeo
```

In order to test our dummy algorithms in this notebook, we load some point clouds:

[2]:
```
epoch1, epoch2 = py4dgeo.read_from_xyz(
    "plane_horizontal_t1.xyz", "plane_horizontal_t2.xyz"
)
corepoints = epoch1.cloud[::100]
```

```
[2024-05-14 13:09:41][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t1.xyz'
[2024-05-14 13:09:41][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t2.xyz'
```

11.2.1 Inherting from the algorithm class

Each algorithm is represented by a class that inherits from `M3C2LikeAlgorithm`. It does not need to inherit directly from that class, but can e.g. also inherit from a more specialized class like `M3C2`. Our first algorithm will behave exactly like `M3C2` only that it reports a different name:

[3]:
```
class RenamedAlgorithm(py4dgeo.M3C2):
    @property
    def name(self):
        return "My super-duper M3C2 algorithm"
```

In the following, we will go over possible customization points for derived algorithm classes.

11.2.2 Changing search directions

Next, we switch to another method of determining the search direction, namely the constant direction `(0, 0, 1)`:

[4]:
```
class DirectionAlgorithm(RenamedAlgorithm):
    def directions(self):
        return np.array([0, 0, 1])
```

[5]:
```
DirectionAlgorithm(
    epochs=(epoch1, epoch2), corepoints=corepoints, cyl_radii=(5.0,)
).run()
```

```
[2024-05-14 13:09:41][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:09:41][INFO] Building KDTree structure with leaf parameter 10
```

[5]:
```
(array([-0.10068844, -0.10017085, -0.10033093, -0.09955525, -0.09876984]),
 array([(0.00187895, 0.00280876, 26, 0.00384368, 24),
        (0.00091541, 0.0024623 , 79, 0.00325635, 75),
        (0.00088682, 0.0027008 , 81, 0.00293255, 75),
        (0.00098606, 0.00300521, 81, 0.00325891, 75),
        (0.00153745, 0.003502  , 37, 0.00306054, 33)],
       dtype=[('lodetection', '<f8'), ('spread1', '<f8'), ('num_samples1', '<i8'), (
↪'spread2', '<f8'), ('num_samples2', '<i8')]))
```

In the above, we chose a constant vector across all corepoints by providing an array of shape `(1x3)`. Alternatively we may provide an array of the same shape as the corepoints array to implement a normal direction that varies for each core point.

11.2.3 Adding Python callbacks to the C++ implementation

py4dgeo implements the M3C2 algorithm in performance-oriented C++. The implementation is substructured as follows: The main algorithm for distance calculation gets passed several callback functions that it calls during distance calculation. For each of these callback functions, there are two implementations:

- An efficient C++ function that is exposed through Python bindings
- A pure Python function that serves as a fallback/reference implementation.

In order to customize the algorithm behaviour, you can also provide your own implementation (either in Python or in C++). See the following example where we like this:

[6]:
```
def my_custom_workingset_finder(*args):
    print("I was called and returned a single point")
    return np.zeros((1, 3))
```

[7]:
```
class CallbackAlgorithm(RenamedAlgorithm):
    def callback_workingset_finder(self):
        return my_custom_workingset_finder
```

[8]:
```
CallbackAlgorithm(
    epochs=(epoch1, epoch2),
    corepoints=corepoints,
    cyl_radii=(2.0,),
    normal_radii=(1.0, 2.0),
).run()
```

```
I was called and returned a single point
I was called and returned a single point
I was called and returned a single point
I was called and returned a single point
I was called and returned a single point
I was called and returned a single point
I was called and returned a single point
I was called and returned a single point
I was called and returned a single point
I was called and returned a single point
```

[8]:
```
(array([0., 0., 0., 0., 0.]),
 array([(nan, nan, 1, nan, 1), (nan, nan, 1, nan, 1),
        (nan, nan, 1, nan, 1), (nan, nan, 1, nan, 1),
        (nan, nan, 1, nan, 1)],
       dtype=[('lodetection', '<f8'), ('spread1', '<f8'), ('num_samples1', '<i8'), (
↪'spread2', '<f8'), ('num_samples2', '<i8')]))
```

In order to learn about what possible callbacks there are and what arguments they are expecting, please have a look at the Python fallback implementations in fallback.py *Note: There will be better documentation about this in the future!* For educational, testing and debugging purposes, there is an implementation of the M3C2 base algorithm that exclusively uses Python fallbacks:

[9]:
```
from py4dgeo.fallback import PythonFallbackM3C2
```

[10]:
```
PythonFallbackM3C2(
    epochs=(epoch1, epoch2),
```

(continues on next page)

(continued from previous page)

```
    corepoints=corepoints,
    cyl_radii=(2.0,),
    normal_radii=(1.0, 2.0),
).run()
```

[10]:
```
(array([-0.10269298, -0.10179986, -0.10091512, -0.09819643, -0.09911222]),
 array([(0.00565384, 0.00417673,  5, 0.00491526,  5),
        (0.00255157, 0.00203436, 11, 0.00380835, 11),
        (0.00281475, 0.00259167, 12, 0.0040656 , 11),
        (0.00295429, 0.0032822 , 11, 0.00377072, 11),
        (0.00360769, 0.00356988, 10, 0.00436149,  9)],
       dtype=[('lodetection', '<f8'), ('spread1', '<f8'), ('num_samples1', '<i8'), (
↪'spread2', '<f8'), ('num_samples2', '<i8')]))
```

Important: Using Python callbacks does considerably slow down your algorithm. While this is true for sequential runs, the effects are even more substantial when applying multi-threading. In the worst case (where you spend all your runtime in Python callbacks), your parallel performance will degrade to sequential. Use this feature for prototyping, but provide a C++ implementation of your callback for production runs.

11.2.4 Other customization

If your algorithm requires a different customization point, please open an issue on the py4dgeo issue tracker.

11.3 Point Cloud Registration

In order to lower the uncertainty of calculations done with `py4dgeo`, point cloud registration should be performed to tightly align the point clouds. `py4dgeo` supports this in two ways:

- It allows you to apply arbitrary affine transformations to epochs and stores the transformation parameters as part of the epoch. You can use the tooling of your choice to calculate these.
- It provides a number of registration algorithms that allow you to calculate the transformation directly in `py4dgeo`. Currently, this is limited to standard ICP.

This notebook show cases both ways of usage.

Note: Be aware that an initial transformation is required that roughly aligns the two point clouds.

[1]:
```
import py4dgeo
import numpy as np
```

We load two epochs and register them to each other:

[2]:
```
epoch1 = py4dgeo.read_from_xyz("plane_horizontal_t1.xyz")  # replace with own data
epoch2 = py4dgeo.read_from_xyz("plane_horizontal_t2.xyz")  # replace with own data
```

```
[2024-05-14 13:13:39][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t1.xyz'
[2024-05-14 13:13:39][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t2.xyz'
```

Registration algorithms use a simple function call interface. It returns an affine transformation as a `3x4` matrix which contains the Rotation matrix $\mathbf{R}$ and the translation vector $\mathbf{t}$:

$$\mathbf{T} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ t_1 & & \\ r_{21} & r_{22} & r_{23} \\ t_2 & & \\ r_{31} & r_{32} & r_{33} \\ t_3 & & \end{pmatrix}$$

When calculating and applying such transformations in `py4dgeo`, it is always possible to specify the *reduction point* $\mathbf{x_0}$. This allows shifting coordinates towards the origin before applying rotations - leading to a numerically robust operation. The algorithm for a transformation is:

$$\mathbf{Tx} = (\mathbf{R}(\mathbf{x} - \mathbf{x_0})) + \mathbf{t} + \mathbf{x_0}$$

[3]:
```
trafo = py4dgeo.iterative_closest_point(
    epoch1, epoch2, reduction_point=np.array([0, 0, 0])
)
```

```
[2024-05-14 13:13:39][INFO] Building KDTree structure with leaf parameter 10
```

This transformation can then be used to transform `epoch2`:

[4]:
```
epoch2.transform(trafo)

## or use an external transformation matrix
# external_trafo = np.loadtxt("trafo.txt") # replace with own data
# epoch2.transform(external_trafo)
```

The `Epoch` class records applied transformations and makes them available through the `transformation` property:

[5]:
```
epoch2.transformation
```

[5]:
```
[Transformation(affine_transformation=array([[ 1.00000000e+00,  6.34089843e-10, -1.
↪42106695e-05,
         -7.59183919e-03],
        [-1.25012920e-10,  9.99999999e-01,  3.58235678e-05,
          7.09922947e-03],
        [ 1.42106695e-05, -3.58235678e-05,  9.99999999e-01,
          2.11030351e+01],
        [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
          1.00000000e+00]]), reduction_point=array([0, 0, 0]))]
```

Finally, we can export the transformed epoch to inspect the registration quality e.g. in CloudCompare.

[6]:
```
# epoch2.save("plane_horizontal_t2_transformed.xyz") # replace with own data
```

11.3.1 Available registration algorithms

Currently only standard point to point ICP is available, but other algorithms are currently implemented:

[7]:
```
?py4dgeo.iterative_closest_point
```

11.4 Basic M3C2-EP algorithm

This presents how the M3C2-EP algorithm (*Winiwarter et al., 2021*) for point cloud distance computation can be run using the `py4dgeo` package.

As a first step, we import the `py4dgeo` and `numpy` packages:

[1]:
```
import numpy as np
import py4dgeo
```

Next, we need to load two datasets that cover the same scene at two different points in time. Point cloud datasets are represented by `numpy` arrays of shape `n x 3` using a 64 bit floating point type (`np.float64`).

Please ensure two datasets include scan positions which are specified by attribute name `sp_name` and scan positions configuration information in `sp_file`.

Here, we work with a rather small data set:

[2]:
```
epoch1, epoch2 = py4dgeo.read_from_las(
    "ahk_2017_652900_5189100_gnd_subarea.laz",
    "ahk_2018A_652900_5189100_gnd_subarea.laz",
    additional_dimensions={"point_source_id": "scanpos_id"},
)
```

```
[2024-05-14 13:09:50][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪ahk_2017_652900_5189100_gnd_subarea.laz'
[2024-05-14 13:09:50][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪ahk_2018A_652900_5189100_gnd_subarea.laz'
```

M3C2-EP leverages knowledge regarding the data acquisition sensor. We extract the sensor orientation details from a JSON configuration file and assign them to a dictionary. These settings are then applied to each epoch of the data since both epochs share the same sensor configuration.

[3]:
```
with open(py4dgeo.find_file("sps.json"), "r") as load_f:
    scanpos_info_dict = eval(load_f.read())
epoch1.scanpos_info = scanpos_info_dict
epoch2.scanpos_info = scanpos_info_dict
```

The analysis of point cloud distances is executed on so-called *core points* (cf. *Lague et al., 2013*). These could be, e.g., one of the input point clouds, a subsampled version thereof, points in an equidistant grid, or something else. Here, we choose a subsampling by taking every 50th point of the reference point cloud:

[4]:
```
corepoints = py4dgeo.read_from_las("ahk_cp_652900_5189100_subarea.laz").cloud
```

```
[2024-05-14 13:09:51][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪ahk_cp_652900_5189100_subarea.laz'
```

The algorithm needs an alignment covariance matrix of shape `12 x 12`, an affine transformation matrix T of shape `3 x 4` and a reduction point $(x_0, y_0, z_0)^T$ (rotation origin, 3 parameters) obtained from aligning the two point clouds.

The transformation follows this notation:

$$T = \begin{pmatrix} t_1 & t_2 & t_3 & t_4 \\ t_5 & t_6 & t_7 & t_8 \\ t_9 & t_{10} & t_{11} & t_{12} \end{pmatrix}$$

Where the transformation is applied as follows:

$$y = \begin{pmatrix} t_1 & t_2 & t_3 \\ t_5 & t_6 & t_7 \\ t_9 & t_{10} & t_{11} \end{pmatrix} \left(x - \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} \right) + \begin{pmatrix} t_4 \\ t_8 \\ t_{12} \end{pmatrix} + \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix}$$

The order of the elements in the covariance matrix is:

$$t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}$$

, meaning that transformation and rotation/scaling parameters are interleaved.

We can decide whether to perform the transformation by a boolean flag 'perform_trans' and it is performed by default.

[5]:
```
Cxx = np.loadtxt(py4dgeo.find_file("Cxx.csv"), dtype=np.float64, delimiter=",")
tfM = np.loadtxt(py4dgeo.find_file("tfM.csv"), dtype=np.float64, delimiter=",")
refPointMov = np.loadtxt(
    py4dgeo.find_file("redPoint.csv"), dtype=np.float64, delimiter=","
)
```

Next, we instantiate the algorithm class and run the distance calculation. The parameters are very similar to the base `M3C2` implementation, but extended to work for M3C2-EP.

[6]:
```
m3c2_ep = py4dgeo.M3C2EP(
    epochs=(epoch1, epoch2),
    corepoints=corepoints,
    normal_radii=(0.5, 1.0, 2.0),
    cyl_radii=(0.5,),
    max_distance=3.0,
    Cxx=Cxx,
    tfM=tfM,
    refPointMov=refPointMov,
)
```

[7]:
```
distances, uncertainties, covariance = m3c2_ep.run()
```

```
M3C2EP running
[2024-05-14 13:09:51][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:09:52][INFO] Building KDTree structure with leaf parameter 10
M3C2EP end
```

The calculated result is an array with one distance per core point. The order of distances corresponds exactly to the order of input core points. In contrast to the base `M3C2`, additional `covariance` information is returned as a third element in the tuple.

[8]:
```
distances[0:8]
```

[8]:
```
array([-0.17164396, -0.62896535, -0.5648335 , -0.15689298, -0.12769975,
       -0.46240109, -0.18725634, -1.08448984])
```

Corresponding to the derived distances, an uncertainty array is returned which contains several quantities that can be accessed individually: The level of detection `lodetection`, the spread of the distance across points in either cloud

(spread1 and spread2, by default measured as the standard deviation of distances) and the total number of points taken into consideration in either cloud (num_samples1 and num_samples2):

[9]:
```
uncertainties["lodetection"][0:8]
```

[9]:
```
array([0.02308354, 0.02319384, 0.02437987, 0.02349072, 0.02414121,
       0.02387164, 0.02578069, 0.03018877])
```

[10]:
```
uncertainties["spread1"][0:8]
```

[10]:
```
array([0.00689291, 0.00695656, 0.00744346, 0.00719902, 0.0074957 ,
       0.00757502, 0.00818117, 0.0084822 ])
```

[11]:
```
uncertainties["num_samples1"][0:8]
```

[11]:
```
array([119, 122, 163,  11, 281,  41, 199, 172])
```

Corresponding to the derived distances, a 3D covariance information for the point cloud is returned.

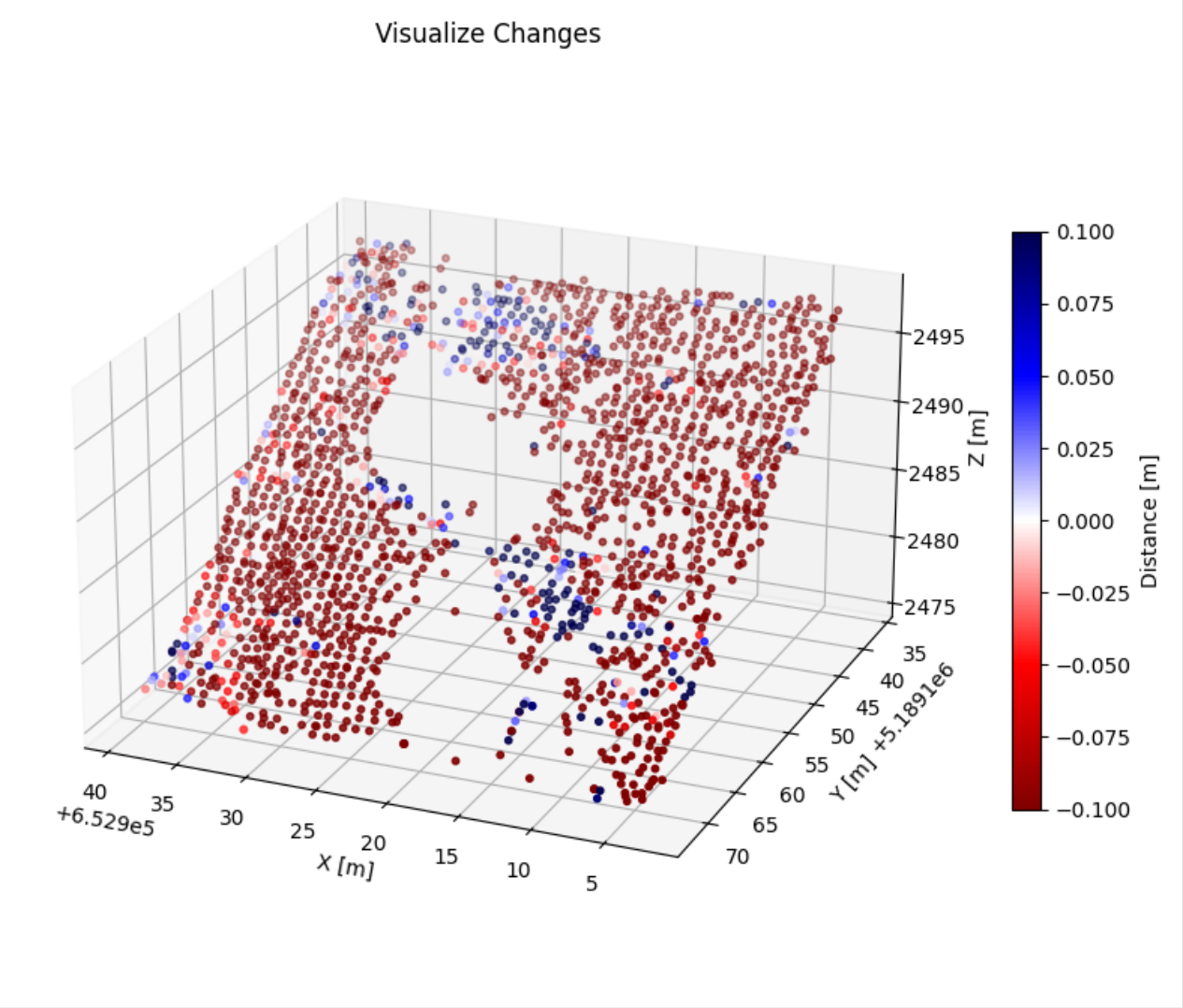
[12]:
```
covariance["cov1"][0, :, :]
```

[12]:
```
array([[ 5.99898020e-05,  1.54621578e-05, -5.85061284e-06],
       [ 1.54621578e-05,  5.78886408e-05, -5.46655748e-06],
       [-5.85061284e-06, -5.46655748e-06,  4.95475375e-05]])
```

Finally we could visualize our distances results.

[13]:
```
import matplotlib.cm as cm
import matplotlib.pyplot as plt


def plt_3d(corepoints, distances):
    fig, ax = plt.subplots(figsize=(10, 10), subplot_kw={"projection": "3d"})

    # add axis labels
    ax.set_xlabel("X [m]")
    ax.set_ylabel("Y [m]")
    ax.set_zlabel("Z [m]")

    # plot the corepoints colored by their distance
    x, y, z = np.transpose(corepoints)
    vmin = np.min(distances)
    vmax = np.max(distances)
    pts = ax.scatter(
        x, y, z, s=10, c=distances, vmin=vmin, vmax=vmax, cmap=cm.seismic_r
    )

    # add colorbar
    cmap = plt.colorbar(pts, shrink=0.5, label="Distance [m]", ax=ax)

    # add title
    ax.set_title("Visualize Changes")

    ax.set_aspect("equal")
    ax.view_init(22, 113)
    plt.show()
```

[14]:
```
plt_3d(corepoints, distances)
```



11.4.1 References

- Winiwarter, L., Anders, K., & Höfle, B. (2021). M3C2-EP: Pushing the limits of 3D topographic point cloud change detection by error propagation. ISPRS Journal of Photogrammetry and Remote Sensing, 178, 240-258. doi: 10.1016/j.isprsjprs.2021.06.011.
- Lague, D., Brodu, N., & Leroux, J. (2013). Accurate 3D comparison of complex topography with terrestrial laser scanner: Application to the Rangitikei canyon (N-Z). ISPRS Journal of Photogrammetry and Remote Sensing, 82, pp. 10-26. doi: 10.1016/j.isprsjprs.2013.04.009.

11.5 4D Objects By Change - Creating an analysis

This notebook explains the data preparation step for the extraction of *4D Objects-By-Change* (4D-OBCs; *Anders et al., 2020*). For details about the method, we refer to the articles by Anders et al. (*2020; 2021*).

[1]:
```
import py4dgeo
import numpy as np
```

The main data structure for the 4D-OBC algorithm is a `SpatiotemporalAnalysis` object. It is always backed by an archive file, which we specify when instantiating the analysis object. If the given filename already exists, we open it, otherwise an empty archive is created:

[2]:
```
analysis = py4dgeo.SpatiotemporalAnalysis("test.zip", force=True)
```

```
[2024-05-14 13:09:26][INFO] Creating analysis file /home/docs/checkouts/readthedocs.org/
↪user_builds/py4dgeo/checkouts/latest/doc/test.zip
```

The analysis object stores all information necessary to (re-)run and extend the analysis. This includes, e.g., the reference epoch, the core points, and the space-time array of change values. However, it does not store all epochs that were used in building up the analysis.

In the following we show how the required components are added to the analysis using the example of the two epochs used for *demonstration of the M3C2*. To be usable during spatiotemporal segmentation, a timestamp is added to these epochs, respectively:

[3]:
```
reference_epoch, epoch = py4dgeo.read_from_xyz(
    "plane_horizontal_t1.xyz", "plane_horizontal_t2.xyz"
)
reference_epoch.timestamp = "March 9th 2022, 16:00"
epoch.timestamp = "March 10th 2022, 16:00"
```

```
[2024-05-14 13:09:26][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t1.xyz'
[2024-05-14 13:09:26][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t2.xyz'
```

The reference epoch needs to be added as such to the analysis:

[4]:
```
analysis.reference_epoch = reference_epoch
```

```
[2024-05-14 13:09:26][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:09:26][INFO] Saving epoch to file '/tmp/tmp8axdc_ev/reference_epoch.zip'
[2024-05-14 13:09:26][INFO] Saving a file without normals.
```

We also need to define the set of core points in use. In this example, we use all points in the reference epoch as core points:

[5]:
```
analysis.corepoints = reference_epoch
```

```
[2024-05-14 13:09:26][INFO] Saving epoch to file '/tmp/tmp09xa9d12/corepoints.zip'
[2024-05-14 13:09:26][INFO] Saving a file without normals.
```

Next, we want to add epochs to the spatiotemporal analysis. For this, we need to define a method that calculates the point cloud distances, for example using the M3C2 algorithm (*Lague et al., 2013*). For details of the algorithm setup, you can have a look at the *M3C2 notebook*.

[6]:
```
analysis.m3c2 = py4dgeo.M3C2(cyl_radii=[2.0], normal_radii=[2.0])
```

Having done all of this, we can add an epoch to the analysis:

[7]:
```
analysis.add_epochs(epoch)
```

```
[2024-05-14 13:09:26][INFO] Removing intermediate results from the analysis file /home/
↪docs/checkouts/readthedocs.org/user_builds/py4dgeo/checkouts/latest/doc/test.zip
[2024-05-14 13:09:26][INFO] Starting: Adding epoch 1/1 to analysis object
[2024-05-14 13:09:26][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:09:26][INFO] Finished in 0.0052s: Adding epoch 1/1 to analysis object
[2024-05-14 13:09:26][INFO] Starting: Rearranging space-time array in memory
[2024-05-14 13:09:26][INFO] Finished in 0.0035s: Rearranging space-time array in memory
[2024-05-14 13:09:26][INFO] Starting: Updating disk-based analysis archive with new␣
↪epochs
[2024-05-14 13:09:26][INFO] Finished in 0.0009s: Updating disk-based analysis archive␣
↪with new epochs
```

Note that the `add_epochs` can be passed several epochs at the same time to save some costly memory operations: `add_epochs(*list_of_epochs)`. Also, `add_epochs` can be called again on existing analysis objects allowing you to update your analysis after additional data acquisition at the same site.

Having done this, we can inspect some of the data contained in the analysis:

[8]:
```
np.set_printoptions(threshold=5)
```

[9]:
```
print(f"Space-time distance array: {analysis.distances}")
```

```
Space-time distance array: [[-0.10269298]
 [-0.10275566]
 [-0.10072999]
 ...
 [-0.10194934]
 [-0.1026113 ]
 [-0.1022107 ]]
```

[10]:
```
print(f"Uncertainty array of M3C2 calculation: {analysis.uncertainties}")
```

```
Uncertainty array of M3C2 calculation: [[(0.00565384, 0.00417673, 5, 0.00491526, 5)]
 [(0.0019427 , 0.00149693, 7, 0.00215317, 7)]
 [(0.00363399, 0.00330837, 8, 0.00406884, 8)]
 ...
 [(0.00299548, 0.00228501, 8, 0.00343241, 7)]
 [(0.00444868, 0.00394557, 8, 0.00473712, 7)]
 [(0.0067821 , 0.00662842, 6, 0.00482219, 5)]]
```

[11]:
```
print(f"Timestamp deltas of analysis: {analysis.timedeltas}")
```

```
Timestamp deltas of analysis: [datetime.timedelta(days=1)]
```

Note that the `add_epochs` can be passed several epochs at the same time to save some costly memory operations. Also, `add_epochs` can be called again on existing analysis objects allowing you to update your analysis after additional data acquisition at the same site.

Sometimes, you will want to apply preprocessing in the form of smoothing to the calculated distance array. You can do so by setting the analysis object's `smoothed_distances` property. When this was set, the 4D-OBC algorithm will

operate on the smoothed data instead of the raw data stored in the analysis object. py4dgeo provides a smoothing implementation using a sliding window approach:

[12]:
```
analysis.smoothed_distances = py4dgeo.temporal_averaging(
    analysis.distances, smoothing_window=24
)
```

```
[2024-05-14 13:09:26][INFO] Starting: Smoothing temporal data
[2024-05-14 13:09:26][INFO] Finished in 0.0002s: Smoothing temporal data
```

11.5.1 References

- Anders, K., Winiwarter, L., Lindenbergh, R., Williams, J. G., Vos, S. E., & Höfle, B. (2020). 4D objects-by-change: Spatiotemporal segmentation of geomorphic surface change from LiDAR time series. ISPRS Journal of Photogrammetry and Remote Sensing, 159, pp. 352-363. doi: 10.1016/j.isprsjprs.2019.11.025.
- Anders, K., Winiwarter, L., Mara, H., Lindenbergh, R., Vos, S. E., & Höfle, B. (2021). Fully automatic spatiotemporal segmentation of 3D LiDAR time series for the extraction of natural surface changes. ISPRS Journal of Photogrammetry and Remote Sensing, 173, pp. 297-308. doi: 10.1016/j.isprsjprs.2021.01.015.
- Lague, D., Brodu, N., & Leroux, J. (2013). Accurate 3D comparison of complex topography with terrestrial laser scanner: Application to the Rangitikei canyon (N-Z). ISPRS Journal of Photogrammetry and Remote Sensing, 82, pp. 10-26. doi: 10.1016/j.isprsjprs.2013.04.009.

11.6 4D Objects By Change - Analysis

This notebook explains how the extraction of *4D Objects By Change* (*Anders et al., 2020*) is run in py4dgeo. For details about the algorithm, we refer to the articles by Anders et al. (2020; 2021).

[1]:
```
import py4dgeo
```

The necessary data for the analysis is stored in an analysis file. There is a dedicated *notebook on creating these analysis files*. In this notebook, we assume that the file already exists and contains a space-time array with corresponding metadata. You can pass an absolute path to the analysis class or have py4dgeo locate files for relative paths:

[2]:
```
analysis = py4dgeo.SpatiotemporalAnalysis("synthetic.zip")
```

If needed, the data can be retrieved from the analysis object. Note that this is a lazy operation, meaning that only explicitly requesting such data will trigger the retrieval from disk into memory:

[3]:
```
analysis.distances
```

[3]:
```
array([[0.00000000e+00, 2.58459742e-07, 5.09761587e-07, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 2.58459742e-07, 5.09761587e-07, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 2.58459742e-07, 5.09761587e-07, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       ...,
       [0.00000000e+00, 2.58459742e-07, 5.09761587e-07, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 2.58459742e-07, 5.09761587e-07, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
```

(continues on next page)

(continued from previous page)

```
       [0.00000000e+00, 2.58459742e-07, 5.09761587e-07, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00]])
```

In case (part of) the analysis was run before, in this case we do not want to reuse it. We remove previous results via `invalidate_results()`. By default, all results are invalidated but we may adjust this to keeping the objects or seeds by setting `seeds=False` or `objects=False`.

[4]:
```
analysis.invalidate_results(seeds=True, objects=True)
```

```
[2024-05-14 13:09:19][INFO] Removing intermediate results from the analysis file /home/
↪docs/.cache/py4dgeo/./synthetic.zip
```

Next, we will construct an algorithm object. You can do so by simply instatiating the `RegionGrowingAlgorithm`:

[5]:
```
algo = py4dgeo.RegionGrowingAlgorithm(
    neighborhood_radius=2.0,
    seed_subsampling=30,
    window_width=6,
    minperiod=3,
    height_threshold=0.05,
)
```

The `neighborhood_radius` parameter for this algorithm is of paramount importance: At each core point, a radius search with this radius is performed to determine which other core points are considered to be local neighbors during region growing. This allows us to perform region growing on a fully unstructured set of core points. The `seed_subsampling` parameter is used to speed up the generation of seed candidates by only investigating every n-th core point (here n=30) for changes in its timeseries. This is only done to improve the usability of this notebook - for real, full analysis you should omit that parameter (using its default of 1).

Further parameters can be set for the change point detection in the time series (cf. *Anders et al., 2020*). The `window_width` defines the width of the sliding temporal window that moves along the signal and determines the discrepancy between the first and the second half of the window (i.e. subsequent time series segments). Change point detection is performed using a C++ reimplementation of the Python library ruptures (*Truong et al., 2018*). The parameter `minperiod` controls the minimum period of a detected change feature to be considered as seed candidate. The `height_threshold` (default: `0.0`) specifies the required magnitude of a dectected change (i.e. unsigned difference between magnitude at start epoch and peak magnitude).

Next, we execute the algorithm on our `analysis` object:

[6]:
```
objects = algo.run(analysis)
```

```
[2024-05-14 13:09:19][INFO] Restoring epoch from file '/tmp/tmp7wmj7pzk/corepoints.zip'
[2024-05-14 13:09:19][INFO] Starting: Find seed candidates in time series
[2024-05-14 13:09:19][INFO] Finished in 0.1728s: Find seed candidates in time series
[2024-05-14 13:09:19][INFO] Starting: Sort seed candidates by priority
[2024-05-14 13:09:19][INFO] Finished in 0.1346s: Sort seed candidates by priority
[2024-05-14 13:09:19][INFO] Starting: Performing region growing on seed candidate 1/70
[2024-05-14 13:09:19][INFO] Finished in 0.1466s: Performing region growing on seed␣
↪candidate 1/70
[2024-05-14 13:09:19][INFO] Starting: Performing region growing on seed candidate 35/70
[2024-05-14 13:09:19][INFO] Finished in 0.0840s: Performing region growing on seed␣
↪candidate 35/70
[2024-05-14 13:09:19][INFO] Starting: Performing region growing on seed candidate 59/70
[2024-05-14 13:09:19][INFO] Finished in 0.1114s: Performing region growing on seed␣
↪candidate 59/70
```

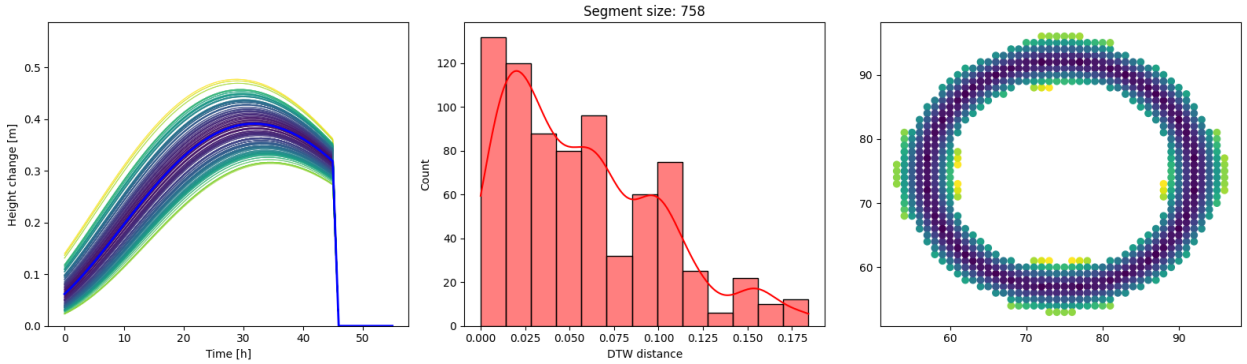
The algorithm returns a list of *4D objects-by-change* (segments in the space time domain):

[7]:
```
print(f"The segmentation extracted {len(objects)} 4D objects-by-change.")
```

```
The segmentation extracted 3 4D objects-by-change.
```

We can plot single objects interactively:

[8]:
```
objects[1].plot()
```



We can check the timespan and location of the seed by accessing the information stored in the object seed properties:

[9]:
```
# Seed coordinate
seed_coord = analysis.corepoints.cloud[objects[1].seed.index]
print(f"Coordinate of seed (XY): {seed_coord[0]:.1f}, {seed_coord[1]:.1f}")

# Seed start and end epoch
print(f"Start epoch: {objects[1].seed.start_epoch}")
print(f"End epoch: {objects[1].seed.end_epoch}")
```

```
Coordinate of seed (XY): 66.0, 90.0
Start epoch: 35
End epoch: 71
```

In this synthetic example there is no reference epoch set. If it were available, we could derive the timestamp of the start and end epochs via the timestamp of the reference epoch in `analysis.reference_epoch.timespamp` and the temporal offset using the start and end epoch id on `analysis.timedeltas`.

Note: Similarly to how the `M3C2` class can be customized by subclassing from it, it is possible to create subclasses of `RegionGrowingAlgorithm` to customize some aspects of the region growing algorithm (not covered in detail in this notebook).

11.6.1 References

- Anders, K., Winiwarter, L., Lindenbergh, R., Williams, J. G., Vos, S. E., & Höfle, B. (2020). 4D objects-by-change: Spatiotemporal segmentation of geomorphic surface change from LiDAR time series. ISPRS Journal of Photogrammetry and Remote Sensing, 159, pp. 352-363. doi: 10.1016/j.isprsjprs.2019.11.025.
- Anders, K., Winiwarter, L., Mara, H., Lindenbergh, R., Vos, S. E., & Höfle, B. (2021). Fully automatic spatiotemporal segmentation of 3D LiDAR time series for the extraction of natural surface changes. ISPRS Journal of Photogrammetry and Remote Sensing, 173, pp. 297-308. doi: 10.1016/j.isprsjprs.2021.01.015.
- Truong, C., Oudre, L., Vayatis, N. (2018): ruptures: Change point detection in python. arXiv preprint: abs/1801.00826.

11.7 4D Object By Change - Customizing the algorithm

The 4D-OBC implementation provided by `py4dgeo` can be customized similarly to *how M3C2 is customized*. The fundamental idea is to create your own algorithm class derived from `py4dgeo`'s `RegionGrowingAlgorithm` class and override only those parts of the algorithm that you want to change. This notebook introduces the currently existing customization points.

[1]:
```
import py4dgeo
import numpy as np

import _py4dgeo  # The C++ bindings module for py4dgeo
```

[2]:
```
# Load test data
analysis = py4dgeo.SpatiotemporalAnalysis("synthetic.zip")
```

[3]:
```
# We are disabling log messages on this tutorial to increase the readability of the␣
↪output
import logging

logging.disable()
```

11.7.1 Distance Measure

By default, 4D-OBC uses a normalized Dynamic Time Warping distance measure. You can provide your own Python function, although all the same warnings as with M3C2 apply: Python code will be significantly slower compared to C++ implementations and will be run sequentially even if you are using OpenMP.

[4]:
```
def custom_distance(params: _py4dgeo.TimeseriesDistanceFunctionData):
    mask = ~np.isnan(params.ts1) & ~np.isnan(params.ts2)
    if not np.any(mask):
        return np.nan

    # Mask the two input arrays
    masked_ts1 = params.ts1[mask]
    masked_ts2 = params.ts2[mask]

    return np.sum(np.abs(masked_ts1 - masked_ts2)) / np.sum(
        np.abs(np.maximum(masked_ts1, masked_ts2))
    )
```

[5]:
```
class CustomDistance4DOBC(py4dgeo.RegionGrowingAlgorithm):
    def distance_measure(self):
        return custom_distance
```

[6]:
```
analysis.invalidate_results()
objects = CustomDistance4DOBC(
    neighborhood_radius=2.0,
    seed_subsampling=20,
).run(analysis)
```

The `params` data structure passed to the distance function contains the following fields: * `ts1` and `ts2` are the time series to compare which may include `NaN` values * `norm1` and `norm2` which are normalization factors

11.7.2 Prioritizing seeds

The 4D-OBC algorithm finds a number of seed locations for region growing and then prioritizes these seeds by sorting them according to a criterion. You can pass your own criterium like this:

[7]:
```
def sorting_criterion(seed):
    # Choose a random score, resulting in random seed order
    return np.random.rand()
```

[8]:
```
class CustomSeedSorting(py4dgeo.RegionGrowingAlgorithm):
    def seed_sorting_scorefunction(self):
        return sorting_criterion
```

[9]:
```
analysis.invalidate_results()
objects = CustomSeedSorting(
    neighborhood_radius=2.0,
    seed_subsampling=20,
).run(analysis)
```

11.7.3 Rejecting grown objects

After the region growing is done, the algorithm class calls it method `filter_objects` to check whether the object should be used or discarded. The method must return `True` (keep) or `False` (discard):

[10]:
```
class RejectSmallObjects(py4dgeo.RegionGrowingAlgorithm):
    def filter_objects(self, obj):
        return len(obj.indices) > 10
```

[11]:
```
analysis.invalidate_results()
objects = RejectSmallObjects(
    neighborhood_radius=2.0,
    seed_subsampling=20,
).run(analysis)
```

11.7.4 Seed point detection

If you would like to run an entirely different algorithm to determine the seeds for region growing, you can do so by overriding `find_seedpoints`:

[12]:
```
from py4dgeo.segmentation import RegionGrowingSeed
```

[13]:
```
class DifferentSeeds(py4dgeo.RegionGrowingAlgorithm):
    def find_seedpoints(self):
        # Use one seed for corepoint 0 and the entire time interval
        return [RegionGrowingSeed(0, 0, self.analysis.distances.shape[1] - 1)]
```

[14]:
```
analysis.invalidate_results()
objects = DifferentSeeds(
    neighborhood_radius=2.0,
```

(continues on next page)

(continued from previous page)

```
    seed_subsampling=20,
).run(analysis)
```

11.8 Correspondence-driven plane-based M3C2 (PBM3C2)

[1]:
```
import py4dgeo
```

In this notebook, we present how the *Correspondence-driven plane-based M3C2* (PB-M3C2, *Zahs et al., 2022*) algorithm for point cloud distance computation using the `py4dgeo` package.

As PB-M3C2 is a learning algorithm, it requires user-labelled input data in the process. This input can either be provided through external tools or be generated using a simple graphical user interface. For the graphical user interface to work best from Jupyter notebooks, we select the `vtk` backend.

[2]:
```
py4dgeo.set_interactive_backend("vtk")
```

We will work on the same demonstrator data we used in the explanation of the *M3C2 algorithm*:

[3]:
```
epoch0, epoch1 = py4dgeo.read_from_xyz(
    "plane_horizontal_t1.xyz", "plane_horizontal_t2.xyz"
)
```

```
[2024-05-14 13:13:10][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t1.xyz'
[2024-05-14 13:13:10][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t2.xyz'
```

Again, we instantiate an instance of the algorithm class. For now, we use only the defaults for its parameters and leave explanation of customization aspects for later.

[4]:
```
alg = py4dgeo.PBM3C2()
```

In a first step, PB-M3C2 will run a plane segmentation algorithm on the provided input point clouds. As a learning algorithm, it then requires user input about corresponding planes. `py4dgeo` offers two ways of doing this: * You can export the segmentation data in XYZ format with four columns: `x`, `y` and `z` of the point cloud, as well as the `segment_id` of the segment the point is associated with. Using that data, you can determine correspondance using your favorite tools or existing workflows. Your input is again expected in a comma-separated text file (CSV). It should contain three columns: The `segment_id` from the first point cloud, the `segment_id` from the second point cloud and a value of `0` or `1` depending on whether the two segments matched. The APIs for this case are shown in this notebook. * You can interactively build the correspondence information in an interactive session. For this, you can call `alg.build_labelled_similarity_features_interactively()`.

Here, we use the first method of using an external tool for labelling. This call will write a total of three files: The above mentioned XYZ files for both epochs, as well as a third file that contains the entire results of the segmentation process. This will allow you to start computation later on without rerunning the segmentation part of the algorithm. You can modify the default file names by passing them to the respective arguments.

[5]:
```
xyz_epoch0, xyz_epoch1, segment_id = alg.export_segmented_point_cloud_and_segments(
    epoch0=epoch0,
    epoch1=epoch1,
)
```

```
[2024-05-14 13:13:10][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:10][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:10][INFO] Transformer Fit
[2024-05-14 13:13:10][INFO] Transformer Transform
[2024-05-14 13:13:10][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:10][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:10][INFO] Transformer Fit
[2024-05-14 13:13:10][INFO] Transformer Transform
[2024-05-14 13:13:10][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:10][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:10][INFO] Transformer Fit
[2024-05-14 13:13:10][INFO] Transformer Transform
[2024-05-14 13:13:10][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:10][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:10][INFO] Transformer Transform
[2024-05-14 13:13:10][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:10][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:10][INFO] Transformer Transform
[2024-05-14 13:13:10][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:10][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:10][INFO] Transformer Fit
[2024-05-14 13:13:10][INFO] Transformer Transform
[2024-05-14 13:13:10][INFO] ----
 The pipeline parameters after restoration are:
{'_Transform_PerPointComputation': PerPointComputation(),
 '_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS'>,
 '_Transform_PerPointComputation__output_file_name': None,
 '_Transform_PerPointComputation__radius': 10,
 '_Transform_PerPointComputation__skip': False,
 '_Transform_Second_Segmentation': Segmentation(with_previously_computed_segments=True),
 '_Transform_Second_Segmentation__angle_diff_threshold': 1,
 '_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 '_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 '_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 '_Transform_Second_Segmentation__llsv_threshold': 1,
 '_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
 '_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 '_Transform_Second_Segmentation__output_file_name': None,
 '_Transform_Second_Segmentation__radius': 2,
 '_Transform_Second_Segmentation__roughness_threshold': 5,
 '_Transform_Second_Segmentation__skip': False,
 '_Transform_Second_Segmentation__with_previously_computed_segments': True,
 '_Transform_Segmentation': Segmentation(),
 '_Transform_Segmentation__angle_diff_threshold': 1,
 '_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_COLUMNS
↪'>,
 '_Transform_Segmentation__distance_3D_threshold': 1.5,
 '_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 '_Transform_Segmentation__llsv_threshold': 1,
 '_Transform_Segmentation__max_nr_points_neighborhood': 100,
 '_Transform_Segmentation__min_nr_points_per_segment': 5,
 '_Transform_Segmentation__output_file_name': None,
```

(continues on next page)

(continued from previous page)

```
 '_Transform_Segmentation__radius': 2,
 '_Transform_Segmentation__roughness_threshold': 5,
 '_Transform_Segmentation__skip': False,
 '_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('_Transform_PerPointComputation', PerPointComputation()),
           ('_Transform_Segmentation', Segmentation()),
           ('_Transform_Second_Segmentation',
            Segmentation(with_previously_computed_segments=True))],
 'verbose': False}
----

[2024-05-14 13:13:10][INFO] ----
 The pipeline parameters after restoration are:
{'_Transform_ExtractSegments': ExtractSegments(),
 '_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 '_Transform_ExtractSegments__output_file_name': None,
 '_Transform_ExtractSegments__skip': False,
 'memory': None,
 'steps': [('_Transform_ExtractSegments', ExtractSegments())],
 'verbose': False}
----
```

After doing the labelling using your preferred method, you can read it into py4dgeo. We pass the previously exported segmentation information and the externally produced CSV file to the traingin procedure. In this test case, we are distributing the labelled data with the test data:

[6]:
```
alg.training(
    extracted_segments_file_name="extracted_segments.seg",
    extended_y_file_name="testdata-labelling.csv",
)
```

```
[2024-05-14 13:13:10][INFO] Reading segments from file '/home/docs/checkouts/readthedocs.
↪org/user_builds/py4dgeo/checkouts/latest/doc/extracted_segments.seg'
[2024-05-14 13:13:10][INFO] Reading tuples of (segment epoch0, segment epoch1, label)␣
↪from file '/home/docs/.cache/py4dgeo/./testdata-labelling.csv'
[2024-05-14 13:13:10][INFO] Fit ClassifierWrapper
```

We have now trained the algorithm using a scikit-learn classifier. By default, this is a random forest tree. We are now ready to compute the distances analoguous to how distances in standard M3C2 are calculated. This will run the prediction with the trained model and derive distance and uncertainty information from the results:

[7]:
```
distances, uncertainties = alg.compute_distances(epoch0=epoch0, epoch1=epoch1)
```

```
[2024-05-14 13:13:11][INFO] PBM3C2.compute_distances(...)
[2024-05-14 13:13:11][INFO] PBM3C2._compute_distances(...)
[2024-05-14 13:13:11][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:11][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:11][INFO] Transformer Fit
[2024-05-14 13:13:11][INFO] Transformer Transform
[2024-05-14 13:13:11][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:11][INFO] Transformer Fit
```

(continues on next page)

(continued from previous page)

```
[2024-05-14 13:13:11][INFO] Transformer Transform
[2024-05-14 13:13:11][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:11][INFO] Transformer Fit
[2024-05-14 13:13:11][INFO] Transformer Transform
[2024-05-14 13:13:11][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:11][INFO] Transformer Transform
[2024-05-14 13:13:11][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:11][INFO] Transformer Transform
[2024-05-14 13:13:11][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:11][INFO] Transformer Fit
[2024-05-14 13:13:11][INFO] Transformer Transform
[2024-05-14 13:13:11][INFO] ----
 The pipeline parameters after restoration are:
{'epoch0_Transform_PerPointComputation': PerPointComputation(),
 'epoch0_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS
↪'>,
 'epoch0_Transform_PerPointComputation__output_file_name': None,
 'epoch0_Transform_PerPointComputation__radius': 10,
 'epoch0_Transform_PerPointComputation__skip': False,
 'epoch0_Transform_Second_Segmentation': Segmentation(with_previously_computed_
↪segments=True),
 'epoch0_Transform_Second_Segmentation__angle_diff_threshold': 1,
 'epoch0_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_
↪CLOUD_COLUMNS'>,
 'epoch0_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 'epoch0_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch0_Transform_Second_Segmentation__llsv_threshold': 1,
 'epoch0_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
 'epoch0_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 'epoch0_Transform_Second_Segmentation__output_file_name': None,
 'epoch0_Transform_Second_Segmentation__radius': 2,
 'epoch0_Transform_Second_Segmentation__roughness_threshold': 5,
 'epoch0_Transform_Second_Segmentation__skip': False,
 'epoch0_Transform_Second_Segmentation__with_previously_computed_segments': True,
 'epoch0_Transform_Segmentation': Segmentation(),
 'epoch0_Transform_Segmentation__angle_diff_threshold': 1,
 'epoch0_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 'epoch0_Transform_Segmentation__distance_3D_threshold': 1.5,
 'epoch0_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch0_Transform_Segmentation__llsv_threshold': 1,
 'epoch0_Transform_Segmentation__max_nr_points_neighborhood': 100,
 'epoch0_Transform_Segmentation__min_nr_points_per_segment': 5,
 'epoch0_Transform_Segmentation__output_file_name': None,
 'epoch0_Transform_Segmentation__radius': 2,
 'epoch0_Transform_Segmentation__roughness_threshold': 5,
 'epoch0_Transform_Segmentation__skip': False,
 'epoch0_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('epoch0_Transform_PerPointComputation', PerPointComputation()),
           ('epoch0_Transform_Segmentation', Segmentation()),
           ('epoch0_Transform_Second_Segmentation',
```

(continues on next page)

(continued from previous page)

```
            Segmentation(with_previously_computed_segments=True))],
 'verbose': False}
----

[2024-05-14 13:13:11][INFO] ----
 The pipeline parameters after restoration are:
{'epoch0_Transform_ExtractSegments': ExtractSegments(),
 'epoch0_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'epoch0_Transform_ExtractSegments__output_file_name': None,
 'epoch0_Transform_ExtractSegments__skip': False,
 'memory': None,
 'steps': [('epoch0_Transform_ExtractSegments', ExtractSegments())],
 'verbose': False}
----

[2024-05-14 13:13:11][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:11][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:11][INFO] Transformer Fit
[2024-05-14 13:13:11][INFO] Transformer Transform
[2024-05-14 13:13:11][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:11][INFO] Transformer Fit
[2024-05-14 13:13:11][INFO] Transformer Transform
[2024-05-14 13:13:11][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:11][INFO] Transformer Fit
[2024-05-14 13:13:11][INFO] Transformer Transform
[2024-05-14 13:13:11][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:11][INFO] Transformer Transform
[2024-05-14 13:13:11][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:11][INFO] Transformer Transform
[2024-05-14 13:13:11][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:11][INFO] Transformer Fit
[2024-05-14 13:13:11][INFO] Transformer Transform
[2024-05-14 13:13:11][INFO] ----
 The pipeline parameters after restoration are:
{'epoch1_Transform_PerPointComputation': PerPointComputation(),
 'epoch1_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS
↪'>,
 'epoch1_Transform_PerPointComputation__output_file_name': None,
 'epoch1_Transform_PerPointComputation__radius': 10,
 'epoch1_Transform_PerPointComputation__skip': False,
 'epoch1_Transform_Second_Segmentation': Segmentation(with_previously_computed_
↪segments=True),
 'epoch1_Transform_Second_Segmentation__angle_diff_threshold': 1,
 'epoch1_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_
↪CLOUD_COLUMNS'>,
 'epoch1_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 'epoch1_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch1_Transform_Second_Segmentation__llsv_threshold': 1,
 'epoch1_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
 'epoch1_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 'epoch1_Transform_Second_Segmentation__output_file_name': None,
 'epoch1_Transform_Second_Segmentation__radius': 2,
```

(continues on next page)

(continued from previous page)

```
 'epoch1_Transform_Second_Segmentation__roughness_threshold': 5,
 'epoch1_Transform_Second_Segmentation__skip': False,
 'epoch1_Transform_Second_Segmentation__with_previously_computed_segments': True,
 'epoch1_Transform_Segmentation': Segmentation(),
 'epoch1_Transform_Segmentation__angle_diff_threshold': 1,
 'epoch1_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 'epoch1_Transform_Segmentation__distance_3D_threshold': 1.5,
 'epoch1_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch1_Transform_Segmentation__llsv_threshold': 1,
 'epoch1_Transform_Segmentation__max_nr_points_neighborhood': 100,
 'epoch1_Transform_Segmentation__min_nr_points_per_segment': 5,
 'epoch1_Transform_Segmentation__output_file_name': None,
 'epoch1_Transform_Segmentation__radius': 2,
 'epoch1_Transform_Segmentation__roughness_threshold': 5,
 'epoch1_Transform_Segmentation__skip': False,
 'epoch1_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('epoch1_Transform_PerPointComputation', PerPointComputation()),
           ('epoch1_Transform_Segmentation', Segmentation()),
           ('epoch1_Transform_Second_Segmentation',
            Segmentation(with_previously_computed_segments=True))],
 'verbose': False}
----

[2024-05-14 13:13:11][INFO] ----
 The pipeline parameters after restoration are:
{'epoch1_Transform_ExtractSegments': ExtractSegments(),
 'epoch1_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'epoch1_Transform_ExtractSegments__output_file_name': None,
 'epoch1_Transform_ExtractSegments__skip': False,
 'memory': None,
 'steps': [('epoch1_Transform_ExtractSegments', ExtractSegments())],
 'verbose': False}
----

[2024-05-14 13:13:11][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:11][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:13][INFO] ----
 The pipeline parameters after restoration are:
{'Classifier': ClassifierWrapper(),
 'Classifier__classifier': RandomForestClassifier(),
 'Classifier__classifier__bootstrap': True,
 'Classifier__classifier__ccp_alpha': 0.0,
 'Classifier__classifier__class_weight': None,
 'Classifier__classifier__criterion': 'gini',
 'Classifier__classifier__max_depth': None,
 'Classifier__classifier__max_features': 'sqrt',
 'Classifier__classifier__max_leaf_nodes': None,
 'Classifier__classifier__max_samples': None,
 'Classifier__classifier__min_impurity_decrease': 0.0,
 'Classifier__classifier__min_samples_leaf': 1,
```

(continues on next page)

(continued from previous page)

```
 'Classifier__classifier__min_samples_split': 2,
 'Classifier__classifier__min_weight_fraction_leaf': 0.0,
 'Classifier__classifier__monotonic_cst': None,
 'Classifier__classifier__n_estimators': 100,
 'Classifier__classifier__n_jobs': None,
 'Classifier__classifier__oob_score': False,
 'Classifier__classifier__random_state': None,
 'Classifier__classifier__verbose': 0,
 'Classifier__classifier__warm_start': False,
 'Classifier__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'Classifier__diff_between_most_similar_2': 0.1,
 'Classifier__neighborhood_search_radius': 3,
 'Classifier__threshold_probability_most_similar': 0.8,
 'memory': None,
 'steps': [('Classifier', ClassifierWrapper())],
 'verbose': False}
----
```

11.8.1 References

- Zahs, V., Winiwarter, L., Anders, K., Williams, J.G., Rutzinger, M. & Höfle, B. (2022): Correspondence-driven plane-based M3C2 for lower uncertainty in 3D topographic change quantification. ISPRS Journal of Photogrammetry and Remote Sensing, 183, pp. 541-559. DOI: 10.1016/j.isprsjprs.2021.11.018.

[ ]:

11.9 Correspondence-driven plane-based M3C2 (PBM3C2) with known segmentation

In this notebook, we are extending the *PB-M3C2 implementation* to work with segmentation information that is already present in the input data. This is useful if you are embedding the calculation into a larger workflow where a segmentation has already been produced.

[1]:
```
import py4dgeo
```

[2]:
```
py4dgeo.set_interactive_backend("vtk")
```

We are reading the two input epochs from XYZ files which contain a total of four columns: X, Y and Z coordinates, as well a segment ID mapping each point to a segment. The `read_from_xyz` functionality allows us to read additional data columns through its `additional_dimensions` parameter. It is expecting a dictionary that maps the column index to a column name.

[3]:
```
epoch0, epoch1 = py4dgeo.read_from_xyz(
    "plane_horizontal_t1_segmented.xyz",
    "plane_horizontal_t2_segmented.xyz",
    additional_dimensions={3: "segment_id"},
    delimiter=",",
)
```

```
[2024-05-14 13:13:24][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t1_segmented.xyz'
[2024-05-14 13:13:25][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t2_segmented.xyz'
```

Again, we instantiate the algorithm. Due to fundamental differences in the algorithm workflow, we are using a separated algorithm class for this use case:

[4]:
```
alg = py4dgeo.PBM3C2WithSegments()
```

Next, we will read the segmented point cloud, which is part of the input epochs, and reconstruct the required segments from it. As a result, we get the same information that we got from the `export_segments_for_labelling` method in the *base PB-M3C2 implementation*. Again, we need to provide labelling and can choose to do so either interactively or with external tools. In contrast to `export_segments_for_labelling`, `reconstruct_post_segmentation_output` only writes one file - the full segmentation information file (which defaults to `extracted_segments.seg`):

[5]:
```
xyz_epoch0, xyz_epoch1, segments = alg.reconstruct_post_segmentation_output(
    epoch0=epoch0,
    epoch1=epoch1,
)
```

```
[2024-05-14 13:13:25][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:25][INFO] Reconstruct post segmentation output using [x, y, z, segment_
↪id] columns from epoch0
[2024-05-14 13:13:25][INFO] Reconstruct post segmentation output using [x, y, z, segment_
↪id] columns from epoch1
[2024-05-14 13:13:25][INFO] Transformer Fit
[2024-05-14 13:13:25][INFO] Transformer Transform
[2024-05-14 13:13:25][INFO] Transformer Fit
[2024-05-14 13:13:25][INFO] Transformer Transform
[2024-05-14 13:13:25][INFO] Transformer Transform
[2024-05-14 13:13:25][INFO] 'Segments' saved in file: extracted_segments.seg
[2024-05-14 13:13:25][INFO] ----
 The pipeline parameters after restoration are:
{'_Transform_ExtractSegments': ExtractSegments(),
 '_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 '_Transform_ExtractSegments__output_file_name': None,
 '_Transform_ExtractSegments__skip': False,
 '_Transform_Post Segmentation': PostPointCloudSegmentation(),
 '_Transform_Post Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 '_Transform_Post Segmentation__compute_normal': True,
 '_Transform_Post Segmentation__output_file_name': None,
 '_Transform_Post Segmentation__skip': False,
 'memory': None,
 'steps': [('_Transform_Post Segmentation', PostPointCloudSegmentation()),
           ('_Transform_ExtractSegments', ExtractSegments())],
 'verbose': False}
----
```

Having completed the labelling process, we read it back in and start the trainging procedure:

[6]:
```
alg.training(
    extracted_segments_file_name="extracted_segments.seg",
    extended_y_file_name="testdata-labelling2.csv",
)
```

```
[2024-05-14 13:13:25][INFO] Reading segments from file '/home/docs/checkouts/readthedocs.
↪org/user_builds/py4dgeo/checkouts/latest/doc/extracted_segments.seg'
[2024-05-14 13:13:25][INFO] Reading tuples of (segment epoch0, segment epoch1, label)␣
↪from file '/home/docs/.cache/py4dgeo/./testdata-labelling2.csv'
[2024-05-14 13:13:25][INFO] Fit ClassifierWrapper
```

[7]:
```
distances, uncertainties = alg.compute_distances(epoch0, epoch1)
```

```
[2024-05-14 13:13:25][INFO] PBM3C2WithSegments.compute_distances(...)
[2024-05-14 13:13:25][INFO] PBM3C2._compute_distances(...)
[2024-05-14 13:13:25][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:25][INFO] Reconstruct post segmentation output using [x, y, z, segment_
↪id] columns from epoch0
[2024-05-14 13:13:25][INFO] Reconstruct post segmentation output using [x, y, z, segment_
↪id] columns from epoch1
[2024-05-14 13:13:25][INFO] Transformer Transform
[2024-05-14 13:13:25][INFO] Transformer Transform
[2024-05-14 13:13:25][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:26][INFO] ----
 The pipeline parameters after restoration are:
{'Classifier': ClassifierWrapper(),
 'Classifier__classifier': RandomForestClassifier(),
 'Classifier__classifier__bootstrap': True,
 'Classifier__classifier__ccp_alpha': 0.0,
 'Classifier__classifier__class_weight': None,
 'Classifier__classifier__criterion': 'gini',
 'Classifier__classifier__max_depth': None,
 'Classifier__classifier__max_features': 'sqrt',
 'Classifier__classifier__max_leaf_nodes': None,
 'Classifier__classifier__max_samples': None,
 'Classifier__classifier__min_impurity_decrease': 0.0,
 'Classifier__classifier__min_samples_leaf': 1,
 'Classifier__classifier__min_samples_split': 2,
 'Classifier__classifier__min_weight_fraction_leaf': 0.0,
 'Classifier__classifier__monotonic_cst': None,
 'Classifier__classifier__n_estimators': 100,
 'Classifier__classifier__n_jobs': None,
 'Classifier__classifier__oob_score': False,
 'Classifier__classifier__random_state': None,
 'Classifier__classifier__verbose': 0,
 'Classifier__classifier__warm_start': False,
 'Classifier__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'Classifier__diff_between_most_similar_2': 0.1,
 'Classifier__neighborhood_search_radius': 3,
 'Classifier__threshold_probability_most_similar': 0.8,
 'Transform_ExtractSegments': ExtractSegments(),
 'Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'Transform_ExtractSegments__output_file_name': None,
```

(continues on next page)

(continued from previous page)

```
 'Transform_ExtractSegments__skip': False,
 'Transform_Post_Segmentation': PostPointCloudSegmentation(),
 'Transform_Post_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 'Transform_Post_Segmentation__compute_normal': True,
 'Transform_Post_Segmentation__output_file_name': None,
 'Transform_Post_Segmentation__skip': False,
 'memory': None,
 'steps': [('Transform_Post_Segmentation', PostPointCloudSegmentation()),
           ('Transform_ExtractSegments', ExtractSegments()),
           ('Classifier', ClassifierWrapper())],
 'verbose': False}
----
```

Note: When comparing distance results between this notebook and the *base algorithm notebook*, you might notice, that results do not necessarily agree even if the given segmentation information is exactly the same as the one computed in the base algorithm. This is due to the reconstruction process in this algorithm being forced to select the segment position (exported as the *core point*) from the segment points instead of reconstructing the correct position from the base algorithm.

[ ]:

11.10 Additional tools for PB-M3C2

In this notebook, we will provide extension to the *PB-M3C2 workflow* that will be occasionally useful based on your application.

11.10.1 Generation of non-correspondent pairs

For best training results, the user should provide both pairs of segments that do correspond to each other, as well as pairs of segments that do not correspond. In manual labelling workflows, it is much easier to produce high quality corresponding pairs that it is to produce non-corresponding pairs. Here, we provide a function that allows you to generate pairs of non-corresponding segments automatically based on heuristic. The general procedure is exactly the same as in *the base workflow* and will not be further explained here.

[1]:
```
import py4dgeo
```

[2]:
```
py4dgeo.set_interactive_backend("vtk")
```

[3]:
```
epoch0, epoch1 = py4dgeo.read_from_xyz(
    "plane_horizontal_t1.xyz", "plane_horizontal_t2.xyz"
)
```

```
[2024-05-14 13:13:31][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t1.xyz'
[2024-05-14 13:13:31][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t2.xyz'
```

[4]:
```
alg = py4dgeo.PBM3C2()
```

[5]:
```
(
    xyz_epoch0,
    xyz_epoch1,
    extracted_segments,
) = alg.export_segmented_point_cloud_and_segments(
    epoch0=epoch0,
    epoch1=epoch1,
)
```

```
[2024-05-14 13:13:31][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:31][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:31][INFO] Transformer Fit
[2024-05-14 13:13:31][INFO] Transformer Transform
[2024-05-14 13:13:31][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:31][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:31][INFO] Transformer Fit
[2024-05-14 13:13:31][INFO] Transformer Transform
[2024-05-14 13:13:31][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:31][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:31][INFO] Transformer Fit
[2024-05-14 13:13:31][INFO] Transformer Transform
[2024-05-14 13:13:31][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:31][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:31][INFO] Transformer Transform
[2024-05-14 13:13:31][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:31][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:31][INFO] Transformer Transform
[2024-05-14 13:13:31][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:31][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:31][INFO] Transformer Fit
[2024-05-14 13:13:31][INFO] Transformer Transform
[2024-05-14 13:13:31][INFO] ----
 The pipeline parameters after restoration are:
{'_Transform_PerPointComputation': PerPointComputation(),
 '_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS'>,
 '_Transform_PerPointComputation__output_file_name': None,
 '_Transform_PerPointComputation__radius': 10,
 '_Transform_PerPointComputation__skip': False,
 '_Transform_Second_Segmentation': Segmentation(with_previously_computed_segments=True),
 '_Transform_Second_Segmentation__angle_diff_threshold': 1,
 '_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 '_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 '_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 '_Transform_Second_Segmentation__llsv_threshold': 1,
 '_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
 '_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 '_Transform_Second_Segmentation__output_file_name': None,
 '_Transform_Second_Segmentation__radius': 2,
 '_Transform_Second_Segmentation__roughness_threshold': 5,
 '_Transform_Second_Segmentation__skip': False,
```

(continues on next page)

(continued from previous page)

```
 '_Transform_Second_Segmentation__with_previously_computed_segments': True,
 '_Transform_Segmentation': Segmentation(),
 '_Transform_Segmentation__angle_diff_threshold': 1,
 '_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_COLUMNS
↪'>,
 '_Transform_Segmentation__distance_3D_threshold': 1.5,
 '_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 '_Transform_Segmentation__llsv_threshold': 1,
 '_Transform_Segmentation__max_nr_points_neighborhood': 100,
 '_Transform_Segmentation__min_nr_points_per_segment': 5,
 '_Transform_Segmentation__output_file_name': None,
 '_Transform_Segmentation__radius': 2,
 '_Transform_Segmentation__roughness_threshold': 5,
 '_Transform_Segmentation__skip': False,
 '_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('_Transform_PerPointComputation', PerPointComputation()),
           ('_Transform_Segmentation', Segmentation()),
           ('_Transform_Second_Segmentation',
            Segmentation(with_previously_computed_segments=True))],
 'verbose': False}
----

[2024-05-14 13:13:31][INFO] ----
 The pipeline parameters after restoration are:
{'_Transform_ExtractSegments': ExtractSegments(),
 '_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 '_Transform_ExtractSegments__output_file_name': None,
 '_Transform_ExtractSegments__skip': False,
 'memory': None,
 'steps': [('_Transform_ExtractSegments', ExtractSegments())],
 'verbose': False}
----
```

Now, we will use labelling data from the file `testdata-labelling-correspondent-only.csv`, which does not contain any pairs of non-corresponding segments. Running `add_no_corresponding_seg` on this data, we automatically generate these. There are two heuristics that can be selected through the `algorithm` parameter: * `random`: For each segment in one epoch, label a random segment from the neighborhood in the other epoch as non-corresponding. * `closes`: For each segment in one epoch, take the closest segment in the other epoch and label it non-corresponding.

The neighborhood of a segment is defined by the threshold parameter given as `threshold_max_distance`.

[6]:
```
augmented_extended_y = py4dgeo.add_no_corresponding_seg(
    segments=extracted_segments,
    threshold_max_distance=5,
    algorithm="random",
    extended_y_file_name="testdata-labelling-correspondent-only.csv",
)
```

```
[2024-05-14 13:13:31][INFO] Reading tuples of (segment epoch0, segment epoch1, label)␣
↪from file '/home/docs/.cache/py4dgeo/./testdata-labelling-correspondent-only.csv'
[2024-05-14 13:13:31][INFO] Building KDTree structure with leaf parameter 10
```

We can then run the training algorithm, passing directly the augmented labelling data:

[7]:
```
alg.training(
    extracted_segments_file_name="extracted_segments.seg",
    extended_y=augmented_extended_y,
)
```

```
[2024-05-14 13:13:31][INFO] Reading segments from file '/home/docs/checkouts/readthedocs.
↪org/user_builds/py4dgeo/checkouts/latest/doc/extracted_segments.seg'
[2024-05-14 13:13:31][INFO] Fit ClassifierWrapper
```

[8]:
```
distances, uncertainties = alg.compute_distances(epoch0=epoch0, epoch1=epoch1)
```

```
[2024-05-14 13:13:32][INFO] PBM3C2.compute_distances(...)
[2024-05-14 13:13:32][INFO] PBM3C2._compute_distances(...)
[2024-05-14 13:13:32][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:32][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:32][INFO] Transformer Fit
[2024-05-14 13:13:32][INFO] Transformer Transform
[2024-05-14 13:13:32][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:32][INFO] Transformer Fit
[2024-05-14 13:13:32][INFO] Transformer Transform
[2024-05-14 13:13:32][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:32][INFO] Transformer Fit
[2024-05-14 13:13:32][INFO] Transformer Transform
[2024-05-14 13:13:32][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:32][INFO] Transformer Transform
[2024-05-14 13:13:32][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:32][INFO] Transformer Transform
[2024-05-14 13:13:32][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:32][INFO] Transformer Fit
[2024-05-14 13:13:32][INFO] Transformer Transform
[2024-05-14 13:13:32][INFO] ----
 The pipeline parameters after restoration are:
{'epoch0_Transform_PerPointComputation': PerPointComputation(),
 'epoch0_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS
↪'>,
 'epoch0_Transform_PerPointComputation__output_file_name': None,
 'epoch0_Transform_PerPointComputation__radius': 10,
 'epoch0_Transform_PerPointComputation__skip': False,
 'epoch0_Transform_Second_Segmentation': Segmentation(with_previously_computed_
↪segments=True),
 'epoch0_Transform_Second_Segmentation__angle_diff_threshold': 1,
 'epoch0_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_
↪CLOUD_COLUMNS'>,
 'epoch0_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 'epoch0_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch0_Transform_Second_Segmentation__llsv_threshold': 1,
 'epoch0_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
 'epoch0_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 'epoch0_Transform_Second_Segmentation__output_file_name': None,
 'epoch0_Transform_Second_Segmentation__radius': 2,
 'epoch0_Transform_Second_Segmentation__roughness_threshold': 5,
 'epoch0_Transform_Second_Segmentation__skip': False,
```

(continues on next page)

(continued from previous page)

```
 'epoch0_Transform_Second_Segmentation__with_previously_computed_segments': True,
 'epoch0_Transform_Segmentation': Segmentation(),
 'epoch0_Transform_Segmentation__angle_diff_threshold': 1,
 'epoch0_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 'epoch0_Transform_Segmentation__distance_3D_threshold': 1.5,
 'epoch0_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch0_Transform_Segmentation__llsv_threshold': 1,
 'epoch0_Transform_Segmentation__max_nr_points_neighborhood': 100,
 'epoch0_Transform_Segmentation__min_nr_points_per_segment': 5,
 'epoch0_Transform_Segmentation__output_file_name': None,
 'epoch0_Transform_Segmentation__radius': 2,
 'epoch0_Transform_Segmentation__roughness_threshold': 5,
 'epoch0_Transform_Segmentation__skip': False,
 'epoch0_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('epoch0_Transform_PerPointComputation', PerPointComputation()),
           ('epoch0_Transform_Segmentation', Segmentation()),
           ('epoch0_Transform_Second_Segmentation',
            Segmentation(with_previously_computed_segments=True))],
 'verbose': False}
----

[2024-05-14 13:13:32][INFO] ----
 The pipeline parameters after restoration are:
{'epoch0_Transform_ExtractSegments': ExtractSegments(),
 'epoch0_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'epoch0_Transform_ExtractSegments__output_file_name': None,
 'epoch0_Transform_ExtractSegments__skip': False,
 'memory': None,
 'steps': [('epoch0_Transform_ExtractSegments', ExtractSegments())],
 'verbose': False}
----

[2024-05-14 13:13:32][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:32][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:32][INFO] Transformer Fit
[2024-05-14 13:13:32][INFO] Transformer Transform
[2024-05-14 13:13:32][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:32][INFO] Transformer Fit
[2024-05-14 13:13:32][INFO] Transformer Transform
[2024-05-14 13:13:32][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:32][INFO] Transformer Fit
[2024-05-14 13:13:32][INFO] Transformer Transform
[2024-05-14 13:13:32][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:32][INFO] Transformer Transform
[2024-05-14 13:13:32][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:32][INFO] Transformer Transform
[2024-05-14 13:13:32][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:32][INFO] Transformer Fit
[2024-05-14 13:13:32][INFO] Transformer Transform
[2024-05-14 13:13:32][INFO] ----
```

(continues on next page)

(continued from previous page)

```
 The pipeline parameters after restoration are:
{'epoch1_Transform_PerPointComputation': PerPointComputation(),
 'epoch1_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS
↪'>,
 'epoch1_Transform_PerPointComputation__output_file_name': None,
 'epoch1_Transform_PerPointComputation__radius': 10,
 'epoch1_Transform_PerPointComputation__skip': False,
 'epoch1_Transform_Second_Segmentation': Segmentation(with_previously_computed_
↪segments=True),
 'epoch1_Transform_Second_Segmentation__angle_diff_threshold': 1,
 'epoch1_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_
↪CLOUD_COLUMNS'>,
 'epoch1_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 'epoch1_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch1_Transform_Second_Segmentation__llsv_threshold': 1,
 'epoch1_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
 'epoch1_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 'epoch1_Transform_Second_Segmentation__output_file_name': None,
 'epoch1_Transform_Second_Segmentation__radius': 2,
 'epoch1_Transform_Second_Segmentation__roughness_threshold': 5,
 'epoch1_Transform_Second_Segmentation__skip': False,
 'epoch1_Transform_Second_Segmentation__with_previously_computed_segments': True,
 'epoch1_Transform_Segmentation': Segmentation(),
 'epoch1_Transform_Segmentation__angle_diff_threshold': 1,
 'epoch1_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 'epoch1_Transform_Segmentation__distance_3D_threshold': 1.5,
 'epoch1_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch1_Transform_Segmentation__llsv_threshold': 1,
 'epoch1_Transform_Segmentation__max_nr_points_neighborhood': 100,
 'epoch1_Transform_Segmentation__min_nr_points_per_segment': 5,
 'epoch1_Transform_Segmentation__output_file_name': None,
 'epoch1_Transform_Segmentation__radius': 2,
 'epoch1_Transform_Segmentation__roughness_threshold': 5,
 'epoch1_Transform_Segmentation__skip': False,
 'epoch1_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('epoch1_Transform_PerPointComputation', PerPointComputation()),
           ('epoch1_Transform_Segmentation', Segmentation()),
           ('epoch1_Transform_Second_Segmentation',
            Segmentation(with_previously_computed_segments=True))],
 'verbose': False}
----

[2024-05-14 13:13:32][INFO] ----
 The pipeline parameters after restoration are:
{'epoch1_Transform_ExtractSegments': ExtractSegments(),
 'epoch1_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'epoch1_Transform_ExtractSegments__output_file_name': None,
 'epoch1_Transform_ExtractSegments__skip': False,
 'memory': None,
 'steps': [('epoch1_Transform_ExtractSegments', ExtractSegments())],
```

(continues on next page)

(continued from previous page)

```
 'verbose': False}
----

[2024-05-14 13:13:32][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:32][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:34][INFO] ----
 The pipeline parameters after restoration are:
{'Classifier': ClassifierWrapper(),
 'Classifier__classifier': RandomForestClassifier(),
 'Classifier__classifier__bootstrap': True,
 'Classifier__classifier__ccp_alpha': 0.0,
 'Classifier__classifier__class_weight': None,
 'Classifier__classifier__criterion': 'gini',
 'Classifier__classifier__max_depth': None,
 'Classifier__classifier__max_features': 'sqrt',
 'Classifier__classifier__max_leaf_nodes': None,
 'Classifier__classifier__max_samples': None,
 'Classifier__classifier__min_impurity_decrease': 0.0,
 'Classifier__classifier__min_samples_leaf': 1,
 'Classifier__classifier__min_samples_split': 2,
 'Classifier__classifier__min_weight_fraction_leaf': 0.0,
 'Classifier__classifier__monotonic_cst': None,
 'Classifier__classifier__n_estimators': 100,
 'Classifier__classifier__n_jobs': None,
 'Classifier__classifier__oob_score': False,
 'Classifier__classifier__random_state': None,
 'Classifier__classifier__verbose': 0,
 'Classifier__classifier__warm_start': False,
 'Classifier__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'Classifier__diff_between_most_similar_2': 0.1,
 'Classifier__neighborhood_search_radius': 3,
 'Classifier__threshold_probability_most_similar': 0.8,
 'memory': None,
 'steps': [('Classifier', ClassifierWrapper())],
 'verbose': False}
----
```

11.11 Applying PB-M3C2 in long term monitoring

In applications where data from the same observation site is acquired over a long period of time, it is desirable to carry out the training of the PB-M3C2 algorithm once and then apply the trained model to newly acquired epochs. This notebook explains how this process is implemented in `py4dgeo`. First, we are carrying out the training procedure like we did in the explanation of the *base workflow*:

[1]:
```
import py4dgeo
```

[2]:
```
py4dgeo.set_interactive_backend("vtk")
```

[3]:
```
epoch0, epoch1 = py4dgeo.read_from_xyz(
    "plane_horizontal_t1.xyz", "plane_horizontal_t2.xyz"
)
```

```
[2024-05-14 13:13:17][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t1.xyz'
[2024-05-14 13:13:17][INFO] Reading point cloud from file '/home/docs/.cache/py4dgeo/./
↪plane_horizontal_t2.xyz'
```

[4]:
```
alg = py4dgeo.PBM3C2()
```

[5]:
```
xyz_epoch0, xyz_epoch1, segment_id = alg.export_segmented_point_cloud_and_segments(
    epoch0=epoch0,
    epoch1=epoch1,
)
```

```
[2024-05-14 13:13:17][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:17][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:17][INFO] Transformer Fit
[2024-05-14 13:13:17][INFO] Transformer Transform
[2024-05-14 13:13:17][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:17][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:17][INFO] Transformer Fit
[2024-05-14 13:13:17][INFO] Transformer Transform
[2024-05-14 13:13:17][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:17][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:17][INFO] Transformer Fit
[2024-05-14 13:13:17][INFO] Transformer Transform
[2024-05-14 13:13:17][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:17][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:18][INFO] Transformer Transform
[2024-05-14 13:13:18][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:18][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:18][INFO] Transformer Transform
[2024-05-14 13:13:18][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:18][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:18][INFO] Transformer Fit
[2024-05-14 13:13:18][INFO] Transformer Transform
[2024-05-14 13:13:18][INFO] ----
 The pipeline parameters after restoration are:
{'_Transform_PerPointComputation': PerPointComputation(),
 '_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS'>,
 '_Transform_PerPointComputation__output_file_name': None,
 '_Transform_PerPointComputation__radius': 10,
 '_Transform_PerPointComputation__skip': False,
 '_Transform_Second_Segmentation': Segmentation(with_previously_computed_segments=True),
 '_Transform_Second_Segmentation__angle_diff_threshold': 1,
 '_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 '_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 '_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 '_Transform_Second_Segmentation__llsv_threshold': 1,
 '_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
```

(continues on next page)

(continued from previous page)

```
 '_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 '_Transform_Second_Segmentation__output_file_name': None,
 '_Transform_Second_Segmentation__radius': 2,
 '_Transform_Second_Segmentation__roughness_threshold': 5,
 '_Transform_Second_Segmentation__skip': False,
 '_Transform_Second_Segmentation__with_previously_computed_segments': True,
 '_Transform_Segmentation': Segmentation(),
 '_Transform_Segmentation__angle_diff_threshold': 1,
 '_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_COLUMNS
↪'>,
 '_Transform_Segmentation__distance_3D_threshold': 1.5,
 '_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 '_Transform_Segmentation__llsv_threshold': 1,
 '_Transform_Segmentation__max_nr_points_neighborhood': 100,
 '_Transform_Segmentation__min_nr_points_per_segment': 5,
 '_Transform_Segmentation__output_file_name': None,
 '_Transform_Segmentation__radius': 2,
 '_Transform_Segmentation__roughness_threshold': 5,
 '_Transform_Segmentation__skip': False,
 '_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('_Transform_PerPointComputation', PerPointComputation()),
           ('_Transform_Segmentation', Segmentation()),
           ('_Transform_Second_Segmentation',
            Segmentation(with_previously_computed_segments=True))],
 'verbose': False}
----

[2024-05-14 13:13:18][INFO] ----
 The pipeline parameters after restoration are:
{'_Transform_ExtractSegments': ExtractSegments(),
 '_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 '_Transform_ExtractSegments__output_file_name': None,
 '_Transform_ExtractSegments__skip': False,
 'memory': None,
 'steps': [('_Transform_ExtractSegments', ExtractSegments())],
 'verbose': False}
----
```

[6]:
```
alg.training(
    extracted_segments_file_name="extracted_segments.seg",
    extended_y_file_name="testdata-labelling.csv",
)
```

```
[2024-05-14 13:13:18][INFO] Reading segments from file '/home/docs/checkouts/readthedocs.
↪org/user_builds/py4dgeo/checkouts/latest/doc/extracted_segments.seg'
[2024-05-14 13:13:18][INFO] Reading tuples of (segment epoch0, segment epoch1, label)␣
↪from file '/home/docs/.cache/py4dgeo/./testdata-labelling.csv'
[2024-05-14 13:13:18][INFO] Fit ClassifierWrapper
```

Having the pre-trained algorithm object `alg`, we would like to save it for reuse in later analysis sessions. We do use Python's `pickle` module for that:

[7]:
```
import pickle
```

[8]:
```
with open("alg.pickle", "wb") as outfile:
    pickle.dump(alg, outfile)
```

Then, in a subsequent session, we can reload the algorithm using `pickle`:

[9]:
```
with open("alg.pickle", "rb") as infile:
    alg = pickle.load(infile)
```

We can then feed new epochs (here, we just use `epoch0` again) into the algorithm. It will apply segmentation on the new epoch and then run the prediction for the new epoch

[10]:
```
(
    _0,
    _1,
    extracted_segments_epoch0,
) = alg.export_segmented_point_cloud_and_segments(
    # must be a new epoch
    epoch0=epoch0,
    # epoch1=None,
    x_y_z_id_epoch0_file_name=None,
    x_y_z_id_epoch1_file_name=None,
    extracted_segments_file_name=None,
)
```

```
[2024-05-14 13:13:18][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:18][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:18][INFO] Transformer Fit
[2024-05-14 13:13:18][INFO] Transformer Transform
[2024-05-14 13:13:18][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:18][INFO] Transformer Fit
[2024-05-14 13:13:18][INFO] Transformer Transform
[2024-05-14 13:13:18][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:18][INFO] Transformer Fit
[2024-05-14 13:13:18][INFO] Transformer Transform
[2024-05-14 13:13:18][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:18][INFO] Transformer Transform
[2024-05-14 13:13:18][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:18][INFO] Transformer Transform
[2024-05-14 13:13:18][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:18][INFO] Transformer Fit
[2024-05-14 13:13:18][INFO] Transformer Transform
[2024-05-14 13:13:18][INFO] ----
 The pipeline parameters after restoration are:
{'_Transform_PerPointComputation': PerPointComputation(),
 '_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS'>,
 '_Transform_PerPointComputation__output_file_name': None,
 '_Transform_PerPointComputation__radius': 10,
 '_Transform_PerPointComputation__skip': False,
 '_Transform_Second_Segmentation': Segmentation(with_previously_computed_segments=True),
 '_Transform_Second_Segmentation__angle_diff_threshold': 1,
 '_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
```

(continues on next page)

(continued from previous page)

```
↪COLUMNS'>,
 '_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 '_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 '_Transform_Second_Segmentation__llsv_threshold': 1,
 '_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
 '_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 '_Transform_Second_Segmentation__output_file_name': None,
 '_Transform_Second_Segmentation__radius': 2,
 '_Transform_Second_Segmentation__roughness_threshold': 5,
 '_Transform_Second_Segmentation__skip': False,
 '_Transform_Second_Segmentation__with_previously_computed_segments': True,
 '_Transform_Segmentation': Segmentation(),
 '_Transform_Segmentation__angle_diff_threshold': 1,
 '_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_COLUMNS
↪'>,
 '_Transform_Segmentation__distance_3D_threshold': 1.5,
 '_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 '_Transform_Segmentation__llsv_threshold': 1,
 '_Transform_Segmentation__max_nr_points_neighborhood': 100,
 '_Transform_Segmentation__min_nr_points_per_segment': 5,
 '_Transform_Segmentation__output_file_name': None,
 '_Transform_Segmentation__radius': 2,
 '_Transform_Segmentation__roughness_threshold': 5,
 '_Transform_Segmentation__skip': False,
 '_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('_Transform_PerPointComputation', PerPointComputation()),
           ('_Transform_Segmentation', Segmentation()),
           ('_Transform_Second_Segmentation',
            Segmentation(with_previously_computed_segments=True))],
 'verbose': False}
----

[2024-05-14 13:13:18][INFO] ----
 The pipeline parameters after restoration are:
{'_Transform_ExtractSegments': ExtractSegments(),
 '_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 '_Transform_ExtractSegments__output_file_name': None,
 '_Transform_ExtractSegments__skip': False,
 'memory': None,
 'steps': [('_Transform_ExtractSegments', ExtractSegments())],
 'verbose': False}
----
```

We can then calculate distances for the new epoch. Note, that in order to disable those parts of the analysis pipeline that are already computed for the reference epoch, we pass the constant dictionary `**py4dgeo.config_epoch0_as_segments`. If you have customized the analysis pipeline, you should adapt the configuration settings accordingly and disable all those steps that are not required for the reference epoch:

[11]:
```
distances, uncertainties = alg.compute_distances(
    epoch0=extracted_segments_epoch0, epoch1=epoch1, **py4dgeo.config_epoch0_as_segments
```

(continues on next page)

(continued from previous page)

```
)
[2024-05-14 13:13:18][INFO] PBM3C2.compute_distances(...)
[2024-05-14 13:13:18][INFO] PBM3C2._compute_distances(...)
[2024-05-14 13:13:18][INFO] ----
 The default parameters are:
 {'epoch0_Transform_ExtractSegments': ExtractSegments(),
 'epoch0_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'epoch0_Transform_ExtractSegments__output_file_name': None,
 'epoch0_Transform_ExtractSegments__skip': False,
 'epoch0_Transform_PerPointComputation': PerPointComputation(),
 'epoch0_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS
↪'>,
 'epoch0_Transform_PerPointComputation__output_file_name': None,
 'epoch0_Transform_PerPointComputation__radius': 10,
 'epoch0_Transform_PerPointComputation__skip': False,
 'epoch0_Transform_Second_Segmentation': Segmentation(with_previously_computed_
↪segments=True),
 'epoch0_Transform_Second_Segmentation__angle_diff_threshold': 1,
 'epoch0_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_
↪CLOUD_COLUMNS'>,
 'epoch0_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 'epoch0_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch0_Transform_Second_Segmentation__llsv_threshold': 1,
 'epoch0_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
 'epoch0_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 'epoch0_Transform_Second_Segmentation__output_file_name': None,
 'epoch0_Transform_Second_Segmentation__radius': 2,
 'epoch0_Transform_Second_Segmentation__roughness_threshold': 5,
 'epoch0_Transform_Second_Segmentation__skip': False,
 'epoch0_Transform_Second_Segmentation__with_previously_computed_segments': True,
 'epoch0_Transform_Segmentation': Segmentation(),
 'epoch0_Transform_Segmentation__angle_diff_threshold': 1,
 'epoch0_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 'epoch0_Transform_Segmentation__distance_3D_threshold': 1.5,
 'epoch0_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch0_Transform_Segmentation__llsv_threshold': 1,
 'epoch0_Transform_Segmentation__max_nr_points_neighborhood': 100,
 'epoch0_Transform_Segmentation__min_nr_points_per_segment': 5,
 'epoch0_Transform_Segmentation__output_file_name': None,
 'epoch0_Transform_Segmentation__radius': 2,
 'epoch0_Transform_Segmentation__roughness_threshold': 5,
 'epoch0_Transform_Segmentation__skip': False,
 'epoch0_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('epoch0_Transform_ExtractSegments', ExtractSegments())],
 'verbose': False}
----

[2024-05-14 13:13:18][INFO] ----
 The pipeline parameters after overwriting are:
```

(continues on next page)

(continued from previous page)

```
{'epoch0_Transform_PerPointComputation': PerPointComputation(skip=True),
 'epoch0_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS
↪'>,
 'epoch0_Transform_PerPointComputation__output_file_name': None,
 'epoch0_Transform_PerPointComputation__radius': 10,
 'epoch0_Transform_PerPointComputation__skip': True,
 'epoch0_Transform_Second_Segmentation': Segmentation(skip=True, with_previously_
↪computed_segments=True),
 'epoch0_Transform_Second_Segmentation__angle_diff_threshold': 1,
 'epoch0_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_
↪CLOUD_COLUMNS'>,
 'epoch0_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 'epoch0_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch0_Transform_Second_Segmentation__llsv_threshold': 1,
 'epoch0_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
 'epoch0_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 'epoch0_Transform_Second_Segmentation__output_file_name': None,
 'epoch0_Transform_Second_Segmentation__radius': 2,
 'epoch0_Transform_Second_Segmentation__roughness_threshold': 5,
 'epoch0_Transform_Second_Segmentation__skip': True,
 'epoch0_Transform_Second_Segmentation__with_previously_computed_segments': True,
 'epoch0_Transform_Segmentation': Segmentation(skip=True),
 'epoch0_Transform_Segmentation__angle_diff_threshold': 1,
 'epoch0_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 'epoch0_Transform_Segmentation__distance_3D_threshold': 1.5,
 'epoch0_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch0_Transform_Segmentation__llsv_threshold': 1,
 'epoch0_Transform_Segmentation__max_nr_points_neighborhood': 100,
 'epoch0_Transform_Segmentation__min_nr_points_per_segment': 5,
 'epoch0_Transform_Segmentation__output_file_name': None,
 'epoch0_Transform_Segmentation__radius': 2,
 'epoch0_Transform_Segmentation__roughness_threshold': 5,
 'epoch0_Transform_Segmentation__skip': True,
 'epoch0_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('epoch0_Transform_PerPointComputation',
            PerPointComputation(skip=True)),
           ('epoch0_Transform_Segmentation', Segmentation(skip=True)),
           ('epoch0_Transform_Second_Segmentation',
            Segmentation(skip=True, with_previously_computed_segments=True))],
 'verbose': False}
----

[2024-05-14 13:13:18][INFO] ----
 The pipeline parameters after overwriting are:
{'epoch0_Transform_ExtractSegments': ExtractSegments(skip=True),
 'epoch0_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'epoch0_Transform_ExtractSegments__output_file_name': None,
 'epoch0_Transform_ExtractSegments__skip': True,
 'memory': None,
 'steps': [('epoch0_Transform_ExtractSegments', ExtractSegments(skip=True))],
```

(continues on next page)

(continued from previous page)

```
 'verbose': False}
----

[2024-05-14 13:13:18][INFO] ----
 The pipeline parameters after restoration are:
{'epoch0_Transform_PerPointComputation': PerPointComputation(),
 'epoch0_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS
↪'>,
 'epoch0_Transform_PerPointComputation__output_file_name': None,
 'epoch0_Transform_PerPointComputation__radius': 10,
 'epoch0_Transform_PerPointComputation__skip': False,
 'epoch0_Transform_Second_Segmentation': Segmentation(with_previously_computed_
↪segments=True),
 'epoch0_Transform_Second_Segmentation__angle_diff_threshold': 1,
 'epoch0_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_
↪CLOUD_COLUMNS'>,
 'epoch0_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 'epoch0_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch0_Transform_Second_Segmentation__llsv_threshold': 1,
 'epoch0_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
 'epoch0_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 'epoch0_Transform_Second_Segmentation__output_file_name': None,
 'epoch0_Transform_Second_Segmentation__radius': 2,
 'epoch0_Transform_Second_Segmentation__roughness_threshold': 5,
 'epoch0_Transform_Second_Segmentation__skip': False,
 'epoch0_Transform_Second_Segmentation__with_previously_computed_segments': True,
 'epoch0_Transform_Segmentation': Segmentation(),
 'epoch0_Transform_Segmentation__angle_diff_threshold': 1,
 'epoch0_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 'epoch0_Transform_Segmentation__distance_3D_threshold': 1.5,
 'epoch0_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch0_Transform_Segmentation__llsv_threshold': 1,
 'epoch0_Transform_Segmentation__max_nr_points_neighborhood': 100,
 'epoch0_Transform_Segmentation__min_nr_points_per_segment': 5,
 'epoch0_Transform_Segmentation__output_file_name': None,
 'epoch0_Transform_Segmentation__radius': 2,
 'epoch0_Transform_Segmentation__roughness_threshold': 5,
 'epoch0_Transform_Segmentation__skip': False,
 'epoch0_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('epoch0_Transform_PerPointComputation', PerPointComputation()),
           ('epoch0_Transform_Segmentation', Segmentation()),
           ('epoch0_Transform_Second_Segmentation',
            Segmentation(with_previously_computed_segments=True))],
 'verbose': False}
----

[2024-05-14 13:13:18][INFO] ----
 The pipeline parameters after restoration are:
{'epoch0_Transform_ExtractSegments': ExtractSegments(),
 'epoch0_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
```

(continues on next page)

(continued from previous page)

```
 'epoch0_Transform_ExtractSegments__output_file_name': None,
 'epoch0_Transform_ExtractSegments__skip': False,
 'memory': None,
 'steps': [('epoch0_Transform_ExtractSegments', ExtractSegments())],
 'verbose': False}
----

[2024-05-14 13:13:18][INFO] ----
 The default parameters are:
 {'epoch1_Transform_ExtractSegments': ExtractSegments(),
 'epoch1_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'epoch1_Transform_ExtractSegments__output_file_name': None,
 'epoch1_Transform_ExtractSegments__skip': False,
 'epoch1_Transform_PerPointComputation': PerPointComputation(),
 'epoch1_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS
↪'>,
 'epoch1_Transform_PerPointComputation__output_file_name': None,
 'epoch1_Transform_PerPointComputation__radius': 10,
 'epoch1_Transform_PerPointComputation__skip': False,
 'epoch1_Transform_Second_Segmentation': Segmentation(with_previously_computed_
↪segments=True),
 'epoch1_Transform_Second_Segmentation__angle_diff_threshold': 1,
 'epoch1_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_
↪CLOUD_COLUMNS'>,
 'epoch1_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 'epoch1_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch1_Transform_Second_Segmentation__llsv_threshold': 1,
 'epoch1_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
 'epoch1_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 'epoch1_Transform_Second_Segmentation__output_file_name': None,
 'epoch1_Transform_Second_Segmentation__radius': 2,
 'epoch1_Transform_Second_Segmentation__roughness_threshold': 5,
 'epoch1_Transform_Second_Segmentation__skip': False,
 'epoch1_Transform_Second_Segmentation__with_previously_computed_segments': True,
 'epoch1_Transform_Segmentation': Segmentation(),
 'epoch1_Transform_Segmentation__angle_diff_threshold': 1,
 'epoch1_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 'epoch1_Transform_Segmentation__distance_3D_threshold': 1.5,
 'epoch1_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch1_Transform_Segmentation__llsv_threshold': 1,
 'epoch1_Transform_Segmentation__max_nr_points_neighborhood': 100,
 'epoch1_Transform_Segmentation__min_nr_points_per_segment': 5,
 'epoch1_Transform_Segmentation__output_file_name': None,
 'epoch1_Transform_Segmentation__radius': 2,
 'epoch1_Transform_Segmentation__roughness_threshold': 5,
 'epoch1_Transform_Segmentation__skip': False,
 'epoch1_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('epoch1_Transform_ExtractSegments', ExtractSegments())],
 'verbose': False}
----
```

(continues on next page)

(continued from previous page)

```
[2024-05-14 13:13:18][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:18][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:18][INFO] Transformer Fit
[2024-05-14 13:13:18][INFO] Transformer Transform
[2024-05-14 13:13:18][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:19][INFO] Transformer Fit
[2024-05-14 13:13:19][INFO] Transformer Transform
[2024-05-14 13:13:19][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:19][INFO] Transformer Fit
[2024-05-14 13:13:19][INFO] Transformer Transform
[2024-05-14 13:13:19][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:19][INFO] Transformer Transform
[2024-05-14 13:13:19][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:19][INFO] Transformer Transform
[2024-05-14 13:13:19][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:19][INFO] Transformer Fit
[2024-05-14 13:13:19][INFO] Transformer Transform
[2024-05-14 13:13:19][INFO] ----
 The pipeline parameters after restoration are:
{'epoch1_Transform_PerPointComputation': PerPointComputation(),
 'epoch1_Transform_PerPointComputation__columns': <class 'py4dgeo.pbm3c2.LLSV_PCA_COLUMNS
↪'>,
 'epoch1_Transform_PerPointComputation__output_file_name': None,
 'epoch1_Transform_PerPointComputation__radius': 10,
 'epoch1_Transform_PerPointComputation__skip': False,
 'epoch1_Transform_Second_Segmentation': Segmentation(with_previously_computed_
↪segments=True),
 'epoch1_Transform_Second_Segmentation__angle_diff_threshold': 1,
 'epoch1_Transform_Second_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_
↪CLOUD_COLUMNS'>,
 'epoch1_Transform_Second_Segmentation__distance_3D_threshold': 1.5,
 'epoch1_Transform_Second_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch1_Transform_Second_Segmentation__llsv_threshold': 1,
 'epoch1_Transform_Second_Segmentation__max_nr_points_neighborhood': 100,
 'epoch1_Transform_Second_Segmentation__min_nr_points_per_segment': 5,
 'epoch1_Transform_Second_Segmentation__output_file_name': None,
 'epoch1_Transform_Second_Segmentation__radius': 2,
 'epoch1_Transform_Second_Segmentation__roughness_threshold': 5,
 'epoch1_Transform_Second_Segmentation__skip': False,
 'epoch1_Transform_Second_Segmentation__with_previously_computed_segments': True,
 'epoch1_Transform_Segmentation': Segmentation(),
 'epoch1_Transform_Segmentation__angle_diff_threshold': 1,
 'epoch1_Transform_Segmentation__columns': <class 'py4dgeo.pbm3c2.SEGMENTED_POINT_CLOUD_
↪COLUMNS'>,
 'epoch1_Transform_Segmentation__distance_3D_threshold': 1.5,
 'epoch1_Transform_Segmentation__distance_orthogonal_threshold': 1.5,
 'epoch1_Transform_Segmentation__llsv_threshold': 1,
 'epoch1_Transform_Segmentation__max_nr_points_neighborhood': 100,
 'epoch1_Transform_Segmentation__min_nr_points_per_segment': 5,
 'epoch1_Transform_Segmentation__output_file_name': None,
 'epoch1_Transform_Segmentation__radius': 2,
```

(continues on next page)

(continued from previous page)

```
 'epoch1_Transform_Segmentation__roughness_threshold': 5,
 'epoch1_Transform_Segmentation__skip': False,
 'epoch1_Transform_Segmentation__with_previously_computed_segments': False,
 'memory': None,
 'steps': [('epoch1_Transform_PerPointComputation', PerPointComputation()),
           ('epoch1_Transform_Segmentation', Segmentation()),
           ('epoch1_Transform_Second_Segmentation',
            Segmentation(with_previously_computed_segments=True))],
 'verbose': False}
----

[2024-05-14 13:13:19][INFO] ----
 The pipeline parameters after restoration are:
{'epoch1_Transform_ExtractSegments': ExtractSegments(),
 'epoch1_Transform_ExtractSegments__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'epoch1_Transform_ExtractSegments__output_file_name': None,
 'epoch1_Transform_ExtractSegments__skip': False,
 'memory': None,
 'steps': [('epoch1_Transform_ExtractSegments', ExtractSegments())],
 'verbose': False}
----

[2024-05-14 13:13:19][INFO] ----
 The default parameters are:
 {'Classifier': ClassifierWrapper(),
 'Classifier__classifier': RandomForestClassifier(),
 'Classifier__classifier__bootstrap': True,
 'Classifier__classifier__ccp_alpha': 0.0,
 'Classifier__classifier__class_weight': None,
 'Classifier__classifier__criterion': 'gini',
 'Classifier__classifier__max_depth': None,
 'Classifier__classifier__max_features': 'sqrt',
 'Classifier__classifier__max_leaf_nodes': None,
 'Classifier__classifier__max_samples': None,
 'Classifier__classifier__min_impurity_decrease': 0.0,
 'Classifier__classifier__min_samples_leaf': 1,
 'Classifier__classifier__min_samples_split': 2,
 'Classifier__classifier__min_weight_fraction_leaf': 0.0,
 'Classifier__classifier__monotonic_cst': None,
 'Classifier__classifier__n_estimators': 100,
 'Classifier__classifier__n_jobs': None,
 'Classifier__classifier__oob_score': False,
 'Classifier__classifier__random_state': None,
 'Classifier__classifier__verbose': 0,
 'Classifier__classifier__warm_start': False,
 'Classifier__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'Classifier__diff_between_most_similar_2': 0.1,
 'Classifier__neighborhood_search_radius': 3,
 'Classifier__threshold_probability_most_similar': 0.8,
 'memory': None,
 'steps': [('Classifier', ClassifierWrapper())],
 'verbose': False}
```

(continues on next page)

(continued from previous page)

```
----

[2024-05-14 13:13:19][INFO] No pipeline parameter is overwritten
[2024-05-14 13:13:19][INFO] Building KDTree structure with leaf parameter 10
[2024-05-14 13:13:20][INFO] ----
 The pipeline parameters after restoration are:
{'Classifier': ClassifierWrapper(),
 'Classifier__classifier': RandomForestClassifier(),
 'Classifier__classifier__bootstrap': True,
 'Classifier__classifier__ccp_alpha': 0.0,
 'Classifier__classifier__class_weight': None,
 'Classifier__classifier__criterion': 'gini',
 'Classifier__classifier__max_depth': None,
 'Classifier__classifier__max_features': 'sqrt',
 'Classifier__classifier__max_leaf_nodes': None,
 'Classifier__classifier__max_samples': None,
 'Classifier__classifier__min_impurity_decrease': 0.0,
 'Classifier__classifier__min_samples_leaf': 1,
 'Classifier__classifier__min_samples_split': 2,
 'Classifier__classifier__min_weight_fraction_leaf': 0.0,
 'Classifier__classifier__monotonic_cst': None,
 'Classifier__classifier__n_estimators': 100,
 'Classifier__classifier__n_jobs': None,
 'Classifier__classifier__oob_score': False,
 'Classifier__classifier__random_state': None,
 'Classifier__classifier__verbose': 0,
 'Classifier__classifier__warm_start': False,
 'Classifier__columns': <class 'py4dgeo.pbm3c2.SEGMENT_COLUMNS'>,
 'Classifier__diff_between_most_similar_2': 0.1,
 'Classifier__neighborhood_search_radius': 3,
 'Classifier__threshold_probability_most_similar': 0.8,
 'memory': None,
 'steps': [('Classifier', ClassifierWrapper())],
 'verbose': False}
----
```

CHAPTER

TWELVE

PYTHON API REFERENCE

12.1 User API reference

This is the complete reference of the Python API for the `py4dgeo` package. It focuses on those aspects relevant to end users that are not interested in algorithm development.

class py4dgeo.**Epoch**(*cloud: ndarray*, *normals: ndarray | None = None*, *additional_dimensions: ndarray | None = None*, *timestamp=None*, *scanpos_info: dict | None = None*)

build_kdtree(*leaf_size=10*, *force_rebuild=False*)

Build the search tree index

Parameters

- **leaf_size** (*int*) – An internal optimization parameter of the search tree data structure. The algorithm uses a bruteforce search on subtrees of size below the given threshold. Increasing this value speeds up search tree build time, but slows down query times.
- **force_rebuild** (*bool*) – Rebuild the search tree even if it was already built before.

calculate_normals(*radius=1.0*, *orientation_vector: ndarray = array([0, 0, 1])*)

Calculate point cloud normals

Parameters

- **radius** – The radius used to determine the neighborhood of a point.
- **orientation_vector** – A vector to determine orientation of the normals. It should point "up".

static load(*filename*)

Construct an Epoch instance by loading it from a file

Parameters
filename (*str*) – The filename to load the epoch from.

property metadata

Provide the metadata of this epoch as a Python dictionary

The return value of this property only makes use of Python built-in data structures such that it can e.g. be serialized using the JSON module. Also, the returned values are understood by Epoch.__init__ such that you can do `Epoch(cloud, **other.metadata)`.

save(*filename*)

Save this epoch to a file

Parameters
filename (*str*) – The filename to save the epoch in.

transform(*transformation: Transformation | None = None*, *affine_transformation: ndarray | None = None*, *rotation: ndarray | None = None*, *translation: ndarray | None = None*, *reduction_point: ndarray | None = None*)

Transform the epoch with an affine transformation

Parameters

- **transformation** (*np.ndarray*) – A Transformation object that describes the transformation to apply. If this argument is given, the other arguments are ignored. This parameter is typically used if the transformation was calculated by py4dgeo itself.
- **affine_transformation** – A 4x4 or 3x4 matrix representing the affine transformation. Given as a numpy array. If this argument is given, the rotation and translation arguments are ignored.
- **rotation** (*np.ndarray*) – A 3x3 matrix specifying the rotation to apply
- **translation** (*np.ndarray*) – A vector specifying the translation to apply
- **reduction_point** (*np.ndarray*) – A translation vector to apply before applying rotation and scaling. This is used to increase the numerical accuracy of transformation. If a transformation is given, this argument is ignored.

property transformation

Access the affine transformations that were applied to this epoch

In order to set this property please use the transform method instead, which will make sure to also apply the transformation.

Returns
Returns a list of applied transformations. These are given as a tuple of a 4x4 matrix defining the affine transformation and the reduction point used when applying it.

py4dgeo.**read_from_las**(**filenames*, *normal_columns=[]*, *additional_dimensions={}*)

Create an epoch from a LAS/LAZ file

Parameters

- **filename** (*str*) – The filename to read from. It is expected to be in LAS/LAZ format and will be processed using laspy.
- **normal_columns** (*list*) – The column names of the normal vector components, e.g. "NormalX", "nx", "normal_x" etc., keep in mind that there must be exactly 3 columns. Leave empty, if your data file does not contain normals.
- **additional_dimensions** (*dict*) – A dictionary, mapping names of additional data dimensions in the input dataset to additional data dimensions in our epoch data structure.

py4dgeo.**read_from_xyz**(**filenames*, *xyz_columns=[0, 1, 2]*, *normal_columns=[]*, *additional_dimensions={}*, ***parse_opts*)

Create an epoch from an xyz file

Parameters

- **filename** (*str*) – The filename to read from. Each line in the input file is expected to contain three space separated numbers.
- **xyz_columns** (*list*) – The column indices of X, Y and Z coordinates. Defaults to [0, 1, 2].

- **normal_columns** (*list*) – The column indices of the normal vector components. Leave empty, if your data file does not contain normals, otherwise exactly three indices for the x, y and z components need to be given.
- **parse_opts** (*dict*) – Additional options forwarded to numpy.genfromtxt. This can be used to e.g. change the delimiter character, remove header_lines or manually specify which columns of the input contain the XYZ coordinates.
- **additional_dimensions** – A dictionary, mapping column indices to names of additional data dimensions. They will be read from the file and are accessible under their names from the created Epoch objects. Additional column indexes start with 3.

py4dgeo.**save_epoch**(*epoch*, *filename*)

Save an epoch to a given filename

Parameters

- **epoch** (Epoch) – The epoch that should be saved.
- **filename** (*str*) – The filename where to save the epoch

py4dgeo.**load_epoch**(*filename*)

Load an epoch from a given filename

Parameters

filename (*str*) – The filename to load the epoch from.

class py4dgeo.**M3C2**(*normal_radii: List[float] | None = None*, *orientation_vector: ndarray = array([0, 0, 1])*, *corepoint_normals: ndarray | None = None*, *cloud_for_normals:* Epoch *| None = None*, ***kwargs*)

Bases: *M3C2LikeAlgorithm*

calculate_distances(*epoch1*, *epoch2*)

Calculate the distances between two epochs

callback_distance_calculation()

The callback used to calculate the distance between two point clouds

callback_workingset_finder()

The callback used to determine the point cloud subset around a corepoint

directions()

The normal direction(s) to use for this algorithm.

run()

Main entry point for running the algorithm

class py4dgeo.**CloudCompareM3C2**(***params*)

Bases: *M3C2*

calculate_distances(*epoch1*, *epoch2*)

Calculate the distances between two epochs

callback_distance_calculation()

The callback used to calculate the distance between two point clouds

callback_workingset_finder()

The callback used to determine the point cloud subset around a corepoint

directions()

The normal direction(s) to use for this algorithm.

run()

Main entry point for running the algorithm

class py4dgeo.**SpatiotemporalAnalysis**(*filename*, *compress=True*, *allow_pickle=True*, *force=False*)

add_epochs(**epochs*)

Add a numbers of epochs to the existing analysis

property corepoints

Access the corepoints of this analysis

property distances

Access the M3C2 distances of this analysis

property distances_for_compute

Retrieve the distance array used for computation

This might be the raw data or smoothed data, based on whether a smoothing was provided by the user.

invalidate_results(*seeds=True*, *objects=True*, *smoothed_distances=False*)

Invalidate (and remove) calculated results

This is automatically called when new epochs are added or when an algorithm sets the `force` option.

property m3c2

Access the M3C2 algorithm of this analysis

property objects

The list of objects by change for this analysis

property reference_epoch

Access the reference epoch of this analysis

property seeds

The list of seed candidates for this analysis

property timedeltas

Access the sequence of time stamp deltas for the time series

property uncertainties

Access the M3C2 uncertainties of this analysis

class py4dgeo.**RegionGrowingAlgorithm**(*seed_subsampling=1*, *seed_candidates=None*, *window_width=24*, *window_min_size=12*, *window_jump=1*, *window_penalty=1.0*, *minperiod=24*, *height_threshold=0.0*, ***kwargs*)

Bases: *RegionGrowingAlgorithmBase*

property analysis

Access the analysis object that the algorithm operates on

This is only available after run has been called.

distance_measure()

Distance measure between two time series

Expected to return a function that accepts two time series and returns the distance.

filter_objects(*obj*)

A filter for objects produced by the region growing algorithm

find_seedpoints()

Calculate seedpoints for the region growing algorithm

run(*analysis*, *force=False*)

Calculate the _segmentation

Parameters

- **analysis** (py4dgeo.segmentation.SpatiotemporalAnalysis) – The analysis object we are working with.
- **force** – Force recalculation of results. If false, some intermediate results will be restored from the analysis object instead of being recalculated.

seed_sorting_scorefunction()

Neighborhood similarity sorting function

py4dgeo.**regular_corepoint_grid**(*lowerleft*, *upperright*, *num_points*, *zval=0.0*)

A helper function to create a regularly spaced grid for the analysis

Parameters

- **lowerleft** (*np.ndarray*) – The lower left corner of the grid. Given as a 2D coordinate.
- **upperright** (*np.ndarray*) – The upper right corner of the grid. Given as a 2D coordinate.
- **num_points** (*tuple*) – A tuple with two entries denoting the number of points to be used in x and y direction
- **zval** (*double*) – The value to fill for the z-direction.

py4dgeo.**set_py4dgeo_logfile**(*filename*)

Set the logfile used by py4dgeo

All log messages produced by py4dgeo are logged into this file in addition to be logged to stdout/stderr. By default, that file is called 'py4dgeo.log'.

Parameters

filename (*str*) – The name of the logfile to use

py4dgeo.**set_memory_policy**(*policy:* MemoryPolicy)

Globally set py4dgeo's memory policy

For details about the memory policy, see *MemoryPolicy*. Use this once before performing any operations. Changing the memory policy in the middle of the computation results in undefined behaviour.

Parameters

policy (MemoryPolicy) – The policy value to globally set

class py4dgeo.**MemoryPolicy**

A descriptor for py4dgeo's memory usage policy

This can be used to describe the memory usage policy that py4dgeo should follow. The implementation of py4dgeo checks the currently set policy whenever it would make a memory allocation of the same order of magnitude as the input pointcloud or the set of corepoints. To globally set the policy, use *set_memory_policy()*.

Currently the following policies are available:

- STRICT: py4dgeo is not allowed to do additional memory allocations. If such an allocation would be required, an error is thrown.

- MINIMAL: py4dgeo is allowed to do additional memory allocations if and only if they are necessary for a seemless operation of the library.
- COREPOINTS: py4dgeo is allowed to do additional memory allocations as part of performance trade-off considerations (e.g. precompute vs. recompute), but only if the allocation is on the order of the number of corepoints. This is the default behaviour of py4dgeo.
- RELAXED: py4dgeo is allowed to do additional memory allocations as part of performance trade-off considerations (e.g. precompute vs. recompute).

py4dgeo.**get_num_threads**()

Get the number of threads currently used by py4dgeo

Returns
The number of threads

Return type
int

py4dgeo.**set_num_threads**(*num_threads: int*)

Set the number of threads to use in py4dgeo

Parameters
num_threads – The number of threads to use

"type num_threads: int

class py4dgeo.**PBM3C2**(*per_point_computation=PerPointComputation()*, *segmentation=Segmentation()*, *second_segmentation=Segmentation()*, *extract_segments=ExtractSegments()*, *classifier=ClassifierWrapper()*)

compute_distances(*epoch0: Epoch | ndarray | None = None*, *epoch1: Epoch | None = None*, *alignment_error: float = 1.1*, ***kwargs*) → Tuple[ndarray, ndarray] | None

Compute the distance between 2 epochs. It also adds the following properties at the end of the computation:
distances, corepoints (corepoints of epoch0), epochs (epoch0, epoch1), uncertainties

Parameters

- **epoch0** – Epoch object. | np.ndarray
- **epoch1** – Epoch object.
- **alignment_error** – alignment error reg between point clouds.
- **kwargs** – Used for customize the default pipeline parameters.

  Getting the default parameters: e.g. "get_pipeline_options"

  In case this parameter is True, the method will print the pipeline options as kwargs.

  e.g. "output_file_name" (of a specific step in the pipeline) default value is "None".
  In case of setting it, the result of computation at that step is dump as xyz file.

  e.g. "distance_3D_threshold" (part of Segmentation Transform)

  this process is stateless

Returns

tuple [distances, uncertainties]
'distances' is np.array (nr_similar_pairs, 1) 'uncertainties' is np.array (nr_similar_pairs,1) and it has the following structure:

dtype=np.dtype(

[
("lodetection", "<f8"), ("spread1", "<f8"), ("num_samples1", "<i8"), ("spread2", "<f8"), ("num_samples2", "<i8"),

]

)

None

`export_segmented_point_cloud_and_segments`(*epoch0: Epoch | None = None*, *epoch1: Epoch | None = None*, *x_y_z_id_epoch0_file_name: str | None = 'x_y_z_id_epoch0.xyz'*, *x_y_z_id_epoch1_file_name: str | None = 'x_y_z_id_epoch1.xyz'*, *extracted_segments_file_name: str | None = 'extracted_segments.seg'*, *concatenate_name=''*, ***kwargs*) → Tuple[ndarray, ndarray | None, ndarray] | None

For each epoch, it returns the segmentation of the point cloud as a numpy array (n_points, 4) and it also serializes them using the provided file names. where each row has the following structure: x, y, z, segment_id

It also generates a numpy array of segments of the form:
X_COLUMN, Y_COLUMN, Z_COLUMN, -> Center of Gravity EPOCH_ID_COLUMN, -> 0/1 EIGENVALUE0_COLUMN, EIGENVALUE1_COLUMN, EIGENVALUE2_COLUMN, EIGENVECTOR_0_X_COLUMN, EIGENVECTOR_0_Y_COLUMN, EIGENVECTOR_0_Z_COLUMN, EIGENVECTOR_1_X_COLUMN, EIGENVECTOR_1_Y_COLUMN, EIGENVECTOR_1_Z_COLUMN, EIGENVECTOR_2_X_COLUMN, EIGENVECTOR_2_Y_COLUMN, EIGENVECTOR_2_Z_COLUMN, -> Normal vector LLSV_COLUMN, -> lowest local surface variation SEGMENT_ID_COLUMN, STANDARD_DEVIATION_COLUMN, NR_POINTS_PER_SEG_COLUMN,

Parameters

- **epoch0** – Epoch object | None
- **epoch1** – Epoch object | None
- **x_y_z_id_epoch0_file_name** – The output file name for epoch0, point cloud segmentation, saved as a numpy array (n_points, 4) (x,y,z, segment_id) | None
- **x_y_z_id_epoch1_file_name** – The output file name for epoch1, point cloud segmentation, saved as a numpy array (n_points, 4) (x,y,z, segment_id) | None
- **extracted_segments_file_name** – The output file name for the file containing the segments, saved as a numpy array containing the column structure introduced above. | None
- **concatenate_name** – String that is utilized to uniquely identify the same transformer between multiple pipelines.

- **kwargs** – Used for customize the default pipeline parameters.

  Getting the default parameters: e.g. "get_pipeline_options"

  In case this parameter is True, the method will print the pipeline options as kwargs.

  e.g. "output_file_name" (of a specific step in the pipeline) default value is "None".
  In case of setting it, the result of computation at that step is dump as xyz file.

  e.g. "distance_3D_threshold" (part of Segmentation Transform)

  this process is stateless

Returns

tuple [x_y_z_id_epoch0, x_y_z_id_epoch1 | None, extracted_segments] | None

generate_extended_labels_interactively(*epoch0: ~py4dgeo.epoch.Epoch | None = None*, *epoch1: ~py4dgeo.epoch.Epoch | None = None*, *builder_extended_y: ~py4dgeo.pbm3c2.BuilderExtended_y_Visually = <py4dgeo.pbm3c2.BuilderExtended_y_Visually object>*, ***kwargs*) → Tuple[ndarray, ndarray] | None

Given 2 Epochs, it builds a pair of (segments and 'extended y').

Parameters

- **epoch0** – Epoch object.
- **epoch1** – Epoch object.
- **builder_extended_y** – The object is used for generating 'extended y', visually.
- **kwargs** – Used for customize the default pipeline parameters.

  Getting the default parameters: e.g. "get_pipeline_options"

  In case this parameter is True, the method will print the pipeline options as kwargs.

  e.g. "output_file_name" (of a specific step in the pipeline) default value is "None".
  In case of setting it, the result of computation at that step is dump as xyz file.

  e.g. "distance_3D_threshold" (part of Segmentation Transform)

  this process is stateless

Returns

tuple [Segments, 'extended y'] | None

where:

'Segments' has the following column structure:
X_COLUMN, Y_COLUMN, Z_COLUMN, -> Center of Gravity EPOCH_ID_COLUMN, -> 0/1 EIGENVALUE0_COLUMN, EIGENVALUE1_COLUMN, EIGENVALUE2_COLUMN, EIGENVECTOR_0_X_COLUMN, EIGENVECTOR_0_Y_COLUMN, EIGENVECTOR_0_Z_COLUMN, EIGENVECTOR_1_X_COLUMN, EIGENVECTOR_1_Y_COLUMN, EIGENVECTOR_1_Z_COLUMN, EIGENVECTOR_2_X_COLUMN, EIGENVECTOR_2_Y_COLUMN, EIGENVEC-

TOR_2_Z_COLUMN, -> Normal vector LLSV_COLUMN, -> lowest local surface variation SEGMENT_ID_COLUMN, STANDARD_DEVIATION_COLUMN, NR_POINTS_PER_SEG_COLUMN,

'extended y' has the following structure: (tuples of index segment from epoch0, index segment from epoch1, label(0/1)) used for learning.

predict(*epoch0:* Epoch | *None* = *None*, *epoch1:* Epoch | *None* = *None*, *epoch_additional_dimensions_lookup: Dict[str, str]* | *None* = *None*, ***kwargs*) → ndarray | None

After extracting the segments from epoch0 and epoch1, it returns a numpy array of corresponding pairs of segments between epoch 0 and epoch 1.

Parameters

- **epoch0** – Epoch object.
- **epoch1** – Epoch object.
- **epoch_additional_dimensions_lookup** – A dictionary that maps between the names of the columns used internally and the names of the columns used by both epoch0 and epoch1.

  No additional column is used.
- **kwargs** – Used for customize the default pipeline parameters.

  Getting the default parameters: e.g. "get_pipeline_options"

  In case this parameter is True, the method will print the pipeline options as kwargs.

  e.g. "output_file_name" (of a specific step in the pipeline) default value is "None".
  In case of setting it, the result of computation at that step is dump as xyz file.

  e.g. "distance_3D_threshold" (part of Segmentation Transform)

  this process is stateless

Returns

A numpy array (n_pairs, segment_features_size*2) where each row contains a pair of segments. | None

training(*segments: ndarray* | *None* = *None*, *extended_y: ndarray* | *None* = *None*, *extracted_segments_file_name: str* = *'extracted_segments.seg'*, *extended_y_file_name: str* = *'extended_y.csv'*) → None

It applies the training algorithm for the input pairs of Segments 'segments' and extended labels 'extended_y'.

Parameters

- **segments** – 'Segments' numpy array of shape (n_segments, segment_size)

  It has the following column structure:
  X_COLUMN, Y_COLUMN, Z_COLUMN, -> Center of Gravity EPOCH_ID_COLUMN, -> 0/1 EIGENVALUE0_COLUMN, EIGENVALUE1_COLUMN, EIGENVALUE2_COLUMN, EIGENVECTOR_0_X_COLUMN, EIGENVECTOR_0_Y_COLUMN, EIGENVECTOR_0_Z_COLUMN, EIGENVECTOR_1_X_COLUMN, EIGENVECTOR_1_Y_COLUMN, EIGENVECTOR_1_Z_COLUMN, EIGENVECTOR_2_X_COLUMN, EIGENVECTOR_2_Y_COLUMN, EIGENVECTOR_2_Z_COLUMN, -> Normal vector LLSV_COLUMN, ->

lowest local surface variation SEGMENT_ID_COLUMN, STANDARD_DEVIATION_COLUMN, NR_POINTS_PER_SEG_COLUMN,

- **extended_y** – numpy array of shape (m_labels, 3) has the following structure: (tuples of index segment from epoch0, index segment from epoch1, label(0/1))
- **extracted_segments_file_name** – In case 'X' is None segments are loaded using 'extracted_segments_file_name'.
- **extended_y_file_name** – In case 'extended_y' is None, this file is used as input fallback.

12.2 Developer API reference

py4dgeo.epoch.**as_epoch**(*cloud*)

Create an epoch from a cloud

Idempotent operation to create an epoch from a cloud.

py4dgeo.epoch.**normalize_timestamp**(*timestamp*)

Bring a given timestamp into a standardized Python format

class py4dgeo.m3c2.**M3C2LikeAlgorithm**(*epochs: Tuple[*Epoch*, ...] | None = None*, *corepoints: ndarray | None = None*, *cyl_radii: List[float] | None = None*, *max_distance: float = 0.0*, *registration_error: float = 0.0*, *robust_aggr: bool = False*)

calculate_distances(*epoch1*, *epoch2*)

Calculate the distances between two epochs

callback_distance_calculation()

The callback used to calculate the distance between two point clouds

callback_workingset_finder()

The callback used to determine the point cloud subset around a corepoint

directions()

The normal direction(s) to use for this algorithm.

run()

Main entry point for running the algorithm

class py4dgeo.fallback.**PythonFallbackM3C2**(*normal_radii: List[float] | None = None*, *orientation_vector: ndarray = array([0, 0, 1])*, *corepoint_normals: ndarray | None = None*, *cloud_for_normals:* Epoch *| None = None*, ***kwargs*)

Bases: *M3C2*

An implementation of M3C2 that makes use of Python fallback implementations

calculate_distances(*epoch1*, *epoch2*)

Calculate the distances between two epochs

callback_distance_calculation()

The callback used to calculate the distance between two point clouds

callback_workingset_finder()

The callback used to determine the point cloud subset around a corepoint

directions()

The normal direction(s) to use for this algorithm.

run()

Main entry point for running the algorithm

class py4dgeo.segmentation.**RegionGrowingAlgorithmBase**(*neighborhood_radius=1.0*, *thresholds=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]*, *min_segments=20*, *max_segments=None*)

property analysis

Access the analysis object that the algorithm operates on

This is only available after run has been called.

distance_measure()

Distance measure between two time series

Expected to return a function that accepts two time series and returns the distance.

filter_objects(*obj*)

A filter for objects produced by the region growing algorithm

Objects are discarded if this method returns False.

find_seedpoints()

Calculate seedpoints for the region growing algorithm

run(*analysis*, *force=False*)

Calculate the _segmentation

Parameters

- **analysis** (py4dgeo.segmentation.SpatiotemporalAnalysis) – The analysis object we are working with.
- **force** – Force recalculation of results. If false, some intermediate results will be restored from the analysis object instead of being recalculated.

seed_sorting_scorefunction()

A function that computes a score for a seed candidate

This function is used to prioritize seed candidates.

class py4dgeo.segmentation.**RegionGrowingSeed**(*index*, *start_epoch*, *end_epoch*)

class py4dgeo.segmentation.**ObjectByChange**(*data*, *seed*, *analysis=None*)

Representation a change object in the spatiotemporal domain

property end_epoch

The index of the end epoch of the change object

property indices

The set of corepoint indices that compose the object by change

plot(*filename=None*)

Create an informative visualization of the Object By Change

Parameters

filename (*str*) – The filename to use to store the plot. Can be omitted to only show plot in a Jupyter notebook session.

property start_epoch

The index of the start epoch of the change object

property threshold

The distance threshold that produced this object

py4dgeo.segmentation.**check_epoch_timestamp**(*epoch*)

Validate an epoch to be used with SpatiotemporalSegmentation

class py4dgeo.util.**Py4DGeoError**(*msg*, *loggername='py4dgeo'*)

py4dgeo.util.**find_file**(*filename*, *fatal=True*)

Find a file of given name on the file system.

This function is intended to use in tests and demo applications to locate data files without resorting to absolute paths. You may use it for your code as well.

It looks in the following locations:

- If an absolute filename is given, it is used
- Check whether the given relative path exists with respect to the current working directory
- Check whether the given relative path exists with respect to the specified XDG data directory (e.g. through the environment variable `XDG_DATA_DIRS`).
- Check whether the given relative path exists in downloaded test data.

Parameters

- **filename** (*str*) – The (relative) filename to search for
- **fatal** (*bool*) – Whether not finding the file should be a fatal error

Returns

An absolute filename

py4dgeo.util.**as_double_precision**(*arr: ndarray*, *policy_check=True*)

Ensure that a numpy array is double precision

This is a no-op if the array is already double precision and makes a copy if it is not. It checks py4dgeo's memory policy before copying.

Parameters

arr (*np.ndarray*) – The numpy array

py4dgeo.util.**make_contiguous**(*arr: ndarray*)

Make a numpy array contiguous

This is a no-op if the array is already contiguous and makes a copy if it is not. It checks py4dgeo's memory policy before copying.

Parameters

arr (*np.ndarray*) – The numpy array

py4dgeo.util.**memory_policy_is_minimum**(*policy:* MemoryPolicy)

Whether or not the globally set memory policy is at least the given one

Parameters

policy (MemoryPolicy) – The policy value to compare against

Returns
Whether the globally set policy is at least the given one

Return type
bool

py4dgeo.util.**append_file_extension**(*filename*, *extension*)

Append a file extension if and only if the original filename has none

py4dgeo.util.**is_iterable**(*obj*)

Whether the object is an iterable (excluding a string)

py4dgeo.logger.**create_default_logger**(*filename=None*)

py4dgeo.logger.**logger_context**(*msg*, *level=20*)

CHAPTER

THIRTEEN

C++ API REFERENCE

This is the complete reference for the (internal) C++ API of `py4dgeo`. You only need to consult this documentation if you are aiming to extend the C++ core of `py4dgeo`.

namespace **py4dgeo**

Typedefs

using **EigenPointCloud** = Eigen::Matrix<double, Eigen::Dynamic, 3, Eigen::RowMajor>

The C++ type for a point cloud.

Point clouds are represented as (nx3) matrices from the Eigen library.

The choice of this type allows us both very efficient implementation of numeric algorithms using the Eigen library, as well as no-copy interoperability with numpy's multidimensional arrays.

using **EigenPointCloudRef** = Eigen::Ref<*EigenPointCloud*>

A non-const reference type for passing around EigenPointCloud.

You should use this in function signatures that accept a point cloud as a parameter and need to modify the point cloud.

using **EigenPointCloudConstRef** = Eigen::Ref<const *EigenPointCloud*>

A const reference type for passing around EigenPointCloud.

You should use this in function signatures that accept a read-only point cloud.

using **EigenNormalSet** = Eigen::Matrix<double, Eigen::Dynamic, 3, Eigen::RowMajor>

The C++ type to represent a set of normal vectors on a point cloud.

In contrast to point clouds, these use double precision.

using **EigenNormalSetRef** = Eigen::Ref<*EigenNormalSet*>

A mutable reference to a set of normal vectors on a point cloud.

using **EigenNormalSetConstRef** = Eigen::Ref<const *EigenNormalSet*>

An immutable reference to a set of normal vectors on a point cloud.

using **IndexType** = Eigen::Index

The type used for point cloud indices.

using **DistanceVector** = std::vector<double>

The variable-sized vector type used for distances.

using **UncertaintyVector** = std::vector<*DistanceUncertainty*>

The variable-sized vector type used for uncertainties.

Enums

enum class **MemoryPolicy**

An enumerator for py4dgeo's memory policy.

This is used and documented through its Python binding equivalent.

Values:

enumerator **STRICT**

enumerator **MINIMAL**

enumerator **COREPOINTS**

enumerator **RELAXED**

struct **DistanceUncertainty**

#include <py4dgeo.hpp> Return structure for the uncertainty of the distance computation.

This structured type is used to describe the uncertainty of point cloud distance at a single corepoint. It contains the level of detection, the spread within both point clouds (e.g. the standard deviation of the distance measure) and the number of sampled points.

Public Members

double **lodetection**

double **spread1**

IndexType **num_samples1**

double **spread2**

IndexType **num_samples2**

namespace **py4dgeo**

class **Epoch**

#include <epoch.hpp> A data structure representing an epoch.

It stores the point cloud itself (without taking ownership of it) and the *KDTree* (with ownership). In the future, relevant metadata fields can be easily added to this data structure without changing any signatures that depend on *Epoch*.

Public Functions

Epoch(const *EigenPointCloudRef* &)

Epoch(std::shared_ptr<*EigenPointCloud*>)

std::ostream &**to_stream**(std::ostream&) const

Public Members

EigenPointCloudRef **cloud**

KDTree **kdtree**

Public Static Functions

static std::unique_ptr<*Epoch*> **from_stream**(std::istream&)

Private Members

std::shared_ptr<*EigenPointCloud*> **owned_cloud**

namespace **py4dgeo**

class **KDTree**

#include <kdtree.hpp> Efficient *KDTree* data structure for nearest neighbor/radius searches.

This data structure allows efficient radius searches in 3D point cloud data. It is based on NanoFLANN: https://github.com/jlblancoc/nanoflann

Public Types

using **RadiusSearchResult** = std::vector<*IndexType*>

Return type used for radius searches.

using **RadiusSearchDistanceResult** = std::vector<std::pair<*IndexType*, double>>

Return type used for radius searches that export calculated distances.

using **NearestNeighborsDistanceResult** = std::vector<std::pair<std::vector<*IndexType*>, std::vector<double>>>

Return type used for nearest neighbor searches.

Public Functions

std::ostream &**saveIndex**(std::ostream &stream) const

Save the index to a (file) stream.

std::istream &**loadIndex**(std::istream &stream)

Load the index from a (file) stream.

void **build_tree**(int leaf)

Build the *KDTree* index.

This initializes the *KDTree* search index. Calling this method is required before performing any nearest neighbors or radius searches.

Parameters

leaf – The threshold parameter defininining at what size the search tree is cutoff. Below the cutoff, a brute force search is performed. This parameter controls a trade off decision between search tree build time and query time.

void **invalidate**()

Invalidate the *KDTree* index.

std::size_t **radius_search**(const double *querypoint, double radius, *RadiusSearchResult* &result) const

Peform radius search around given query point.

This method determines all the points from the point cloud within the given radius of the query point. It returns only the indices and the result is not sorted according to distance.

Parameters

- **querypoint** – **[in]** A pointer to the 3D coordinate of the query point
- **radius** – **[in]** The radius to search within
- **result** – **[out]** A data structure to hold the result. It will be cleared during application.

Returns

The amount of points in the return set

std::size_t **radius_search_with_distances**(const double *querypoint, double radius, *RadiusSearchDistanceResult* &result) const

Perform radius search around given query point exporting distance information.

This method determines all the points from the point cloud within the given radius of the query point. It returns their indices and their distances from the query point. The result is sorted by ascending distance from the query point.

Parameters

- **querypoint** – **[in]** A pointer to the 3D coordinate of the query point
- **radius** – **[in]** The radius to search within
- **result** – **[out]** A data structure to hold the result. It will be cleared during application.

Returns

The amount of points in the return set

void **nearest_neighbors_with_distances**(*EigenPointCloudConstRef* cloud, *NearestNeighborsDistanceResult* &result, int k) const

Calculate the nearest neighbors for an entire point cloud.

Parameters

- **cloud** – **[in]** The point cloud to use as query points
- **result** – **[out]** The indexes and distances of k nearest neighbors for each point
- **k** – **[in]** The amount of nearest neighbors to calculate

int **get_leaf_parameter**() const

Return the leaf parameter this *KDTree* has been built with.

Public Static Functions

static *KDTree* **create**(const *EigenPointCloudRef* &cloud)

Construct instance of *KDTree* from a given point cloud.

This is implemented as a static function instead of a public constructor to ease the implementation of Python bindings.

Parameters

cloud – The point cloud to construct the search tree for

Private Types

using **KDTreeImpl** = nanoflann::KDTreeSingleIndexAdaptor<nanoflann::L2_Simple_Adaptor<double, *Adaptor*, double>, *Adaptor*, 3, *IndexType*>

The NanoFLANN index implementation that we use.

Private Functions

KDTree(const *EigenPointCloudRef* &)

Private constructor from pointcloud - use through *KDTree::create*.

Private Members

friend Epoch

Adaptor **adaptor**

std::shared_ptr<*KDTreeImpl*> **search**

int **leaf_parameter** = 0

struct **Adaptor**

An adaptor between our Eigen data structures and NanoFLANN.

Public Functions

inline std::size_t **kdtree_get_point_count**() const

inline double **kdtree_get_pt**(const *IndexType* idx, const *IndexType* dim) const

template<class **BBOX**>
inline bool **kdtree_get_bbox**(*BBOX*&) const

Public Members

EigenPointCloudRef **cloud**

struct **NoDistancesReturnSet**

A structure to perform efficient radius searches with NanoFLANN.

The built-in return set of NanoFLANN does automatically export the distances as well, which we want to omit if we already know that we do not need the distance information.

Public Functions

inline std::size_t **size**() const

inline bool **full**() const

inline bool **addPoint**(double dist, *IndexType* idx)

inline double **worstDist**() const

Public Members

double **radius**

RadiusSearchResult &**indices**

void *py4dgeo*::**compute_distances**(*EigenPointCloudConstRef*, double, const *Epoch*&, const *Epoch*&, *EigenNormalSetConstRef*, double, double, *DistanceVector*&, *UncertaintyVector*&, const *WorkingSetFinderCallback*&, const *DistanceUncertaintyCalculationCallback*&)

Compute M3C2 distances.

void *py4dgeo*::**compute_multiscale_directions**(const *Epoch*&, *EigenPointCloudConstRef*, const std::vector<double>&, *EigenNormalSetConstRef*, *EigenNormalSetRef*)

Compute M3C2 multi scale directions.

namespace **py4dgeo**

Typedefs

using **EigenSpatiotemporalArray** = Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>

The type to use for the spatiotemporal distance array.

using **EigenSpatiotemporalArrayRef** = Eigen::Ref<*EigenSpatiotemporalArray*>

using **EigenSpatiotemporalArrayConstRef** = Eigen::Ref<const *EigenSpatiotemporalArray*>

using **EigenTimeSeries** = Eigen::Vector<double, Eigen::Dynamic>

The type to use for a TimeSeries.

using **EigenTimeSeriesRef** = Eigen::Ref<*EigenTimeSeries*>

using **EigenTimeSeriesConstRef** = Eigen::Ref<const *EigenTimeSeries*>

using **TimeseriesDistanceFunction** = std::function<double(const *TimeseriesDistanceFunctionData*&)>

The signature to use for a distance function.

struct **TimeseriesDistanceFunctionData**

#include <segmentation.hpp> The data object passed to time series distance functions.

Public Members

EigenTimeSeriesConstRef **ts1**

EigenTimeSeriesConstRef **ts2**

double **norm1**

double **norm2**

struct **ObjectByChange**

#include <segmentation.hpp> Basic data structure for 4D change object.

Public Members

std::unordered_map<*IndexType*, double> **indices_distances**

IndexType **start_epoch**

IndexType **end_epoch**

double **threshold**

struct **RegionGrowingSeed**

Public Members

IndexType **index**

IndexType **start_epoch**

IndexType **end_epoch**

struct **RegionGrowingAlgorithmData**

Public Members

EigenSpatiotemporalArrayConstRef **distances**

const *Epoch* &**corepoints**

double **radius**

RegionGrowingSeed **seed**

std::vector<double> **thresholds**

std::size_t **min_segments**

std::size_t **max_segments**

struct **ChangePointDetectionData**

Public Members

EigenTimeSeriesConstRef **ts**

IndexType **window_width**

IndexType **min_size**

IndexType **jump**

double **penalty**

class **CallbackExceptionVault**

#include <openmp.hpp> A container to handle exceptions in parallel regions.

OpenMP does have limited support for C++ exceptions in parallel regions: Exceptions need to be catched on the same thread they have been thrown on. This class allows to store the first thrown exception to then rethrow it after we left the parallel region. This is a necessary construct to propagate exceptions from Python callbacks through the multithreaded C++ layer back to the calling Python scope. Inspiration is taken from: https://stackoverflow.com/questions/11828539/elegant-exceptionhandling-in-openmp

Public Functions

template<typename **Function**, typename ...**Parameters**>
inline void **run**(*Function* &&f, *Parameters*&&... parameters)

inline void **rethrow**() const

Private Members

std::exception_ptr **ptr** = nullptr

CHAPTER

FOURTEEN

CALLBACK REFERENCE

As described in the custumization tutorial, `py4dgeo` uses a callback software architecture to allow to flexibly change components of the core algorithm while maintaining its performance. Callbacks can be implemented in Python (for rapid prototyping) or C++ (for performance). In this section, we summarize the available types of callbacks and their available implementations.

14.1 Distance + Uncertainty Calculation

This callback is responsible for calculating the distance measure and its uncertainty at one core point. The C++ signature for this callback is the following:

using *py4dgeo*::**DistanceUncertaintyCalculationCallback** = std::function<std::tuple<double, *DistanceUncertainty*>(const DistanceUncertaintyCalculationParameters&)>

The callback type for calculating the distance between two point clouds.

The default implementation calculates the mean and standard deviation:

std::tuple<double, *DistanceUncertainty*> *py4dgeo*::**mean_stddev_distance**(const DistanceUncertaintyCalculationParameters&)

Mean-based implementation of point cloud distance.

This is the default implementation of point cloud distance that takes the mean of both point clouds (center of mass), projects it onto the cylinder axis and calculates the distance.

py4dgeo.fallback.**mean_stddev_distance**(*params: DistanceUncertaintyCalculationParameters*) → tuple

Alternative, a distance measure based on the median and interquartile range is available:

std::tuple<double, *DistanceUncertainty*> *py4dgeo*::**median_iqr_distance**(const DistanceUncertaintyCalculationParameters&)

Median-based implementation of point cloud distance.

Use median of distances in pointcloud instead of mean. This results in a more expensive but more robust distance measure.

py4dgeo.fallback.**median_iqr_distance**(*params: DistanceUncertaintyCalculationParameters*) → tuple

14.2 Working Set Finder

This callback determines which points from a given epoch that are located around a given corepoint should be taken into consideration by the M3C2 algorithm. The C++ signature is the following:

using *py4dgeo*::**WorkingSetFinderCallback** = std::function<*EigenPointCloud*(const WorkingSetFinderParameters&)>

The callback type that determines the point cloud working subset in the vicinity of a core point.

The available implementations perform a radius and a cylinder search:

EigenPointCloud *py4dgeo*::**radius_workingset_finder**(const WorkingSetFinderParameters&)

Implementation of working set finder that performs a regular radius search.

py4dgeo.fallback.**radius_workingset_finder**(*params: WorkingSetFinderParameters*) → ndarray

EigenPointCloud *py4dgeo*::**cylinder_workingset_finder**(const WorkingSetFinderParameters&)

Implementation of a working set finder that performs a cylinder search.

py4dgeo.fallback.**cylinder_workingset_finder**(*params: WorkingSetFinderParameters*) → ndarray

CHAPTER

FIFTEEN

FREQUENTLY ASKED QUESTIONS

15.1 py4dgeo requires more RAM than e.g. CloudCompare/I run out of memory processing a pointcloud that CloudCompare can process on the same computer

This might be related to a deliberate choice in the design of `py4dgeo`: Point clouds are stored as double precision values (`np.float64` or C++'s `double`), while many other software packages that process point clouds (e.g. CloudCompare) go for a mixed precision approach, where the point cloud is stored in single precision and computational results (e.g. M3C2 distances) are represented in double precision. In order to store point clouds in single precision without significant loss of precision, they need to be correctly shifted close to the origin of the space and the resulting offset needs to be taken into account in a lot of computations. With `py4dgeo` being an open system for users to develop their own algorithms, we felt that the risk of caveats related to this transformation process is quite high and therefore decided to choose the very robust (but memory-inefficient) method of storing point clouds in global coordinates as double precision values. Other than what was just explained, the design of the `py4dgeo` library aims for maximum RAM efficiency.

INDEX

E

F

G

I

L

M

N

O

P

R

S

T

U