# Noise Planet

***Release 0.1***

**Sep 29, 2022**

# Getting Started

CHAPTER 1

# Noise Planet

## 1.1 Overview

**NoisePlanet** is a python package for map matching and mapping GeoJson tracks. This library is a project within the research center UMR-AE/CNRS, working on NoisePlanet for noise mapping. Made in collaboration with the *École Nationale des Sciences Géographiques.*

**NoisePlanet** package steps in first in the correction processing. Even though the package is still under improvement, it provides basic correction and mapping visualization.

**Citation :** Dujardin, A., Mermet, S. (2020). État de l'art et suggestions pour la cartographie des données acoustiques mobiles. *Projet de recherche.*

## 1.2 Study

The Noise Planet platform can be found at : http://noise-planet.org/.

Our study was focused and tested on Lyon. The dataset used is visible at : http://noise-planet.org/map_noisecapture/index.html#15/45.7578/4.8320/ and downloaded at : https://dashboard.noise-planet.org/public/question/52ee2bde-2d28-4377-bcbb-061d7cbfa343

CHAPTER 2

# Getting Started

## 2.1 Dependencies

**Packages**

- **numpy,**
- **pandas,**
- **json,**
- **osmnx,**
- **leuvenmapmatching,** *KU Leuven - DTAI Research Group, Sirris - Elucidata Group.*

**Optional packages**

- **matplotlib,**
- **folium,**
- **sqlite3.**

Note that these packages are optional if you don't want to visualize the resulting maps. SQLite3 is used to stock all the informations of a geojson tracks or polygon into an SQL database.

## 2.2 Installation

To install, use :

```
pip install noiseplanet
```

If this doesn't work, clone the repository, and in the noiseplanet folder, use :

```
pip install .
```

## 2.3 Development

If you want to participate to the improvement of this project, clone the repository and open it as a project. We used spyder to create the packages and modules.

## 2.4 Structure

**NoisePlanet** is composed by internal sub-packages:

- **matcher** lets you correct tracks and match it to the Open Street Map network,
- **utils** mainly handles conversion from geojson, metadata etc. to DataFrame,
- **io** handles reading and writing files,
- **db** lets you access a SQLite3 database.
- **ui** is used to generate Leaflet maps,

CHAPTER 3

# Example

## 3.1 Map Matching

The noiseplanet package provides different tools for matching a track to the Open Street Map network.

- matching to the **nearest edge**,
- **hmm** based matching. To match a track, composed by latitudes and longitudes, use :

Firt, import the following packages :

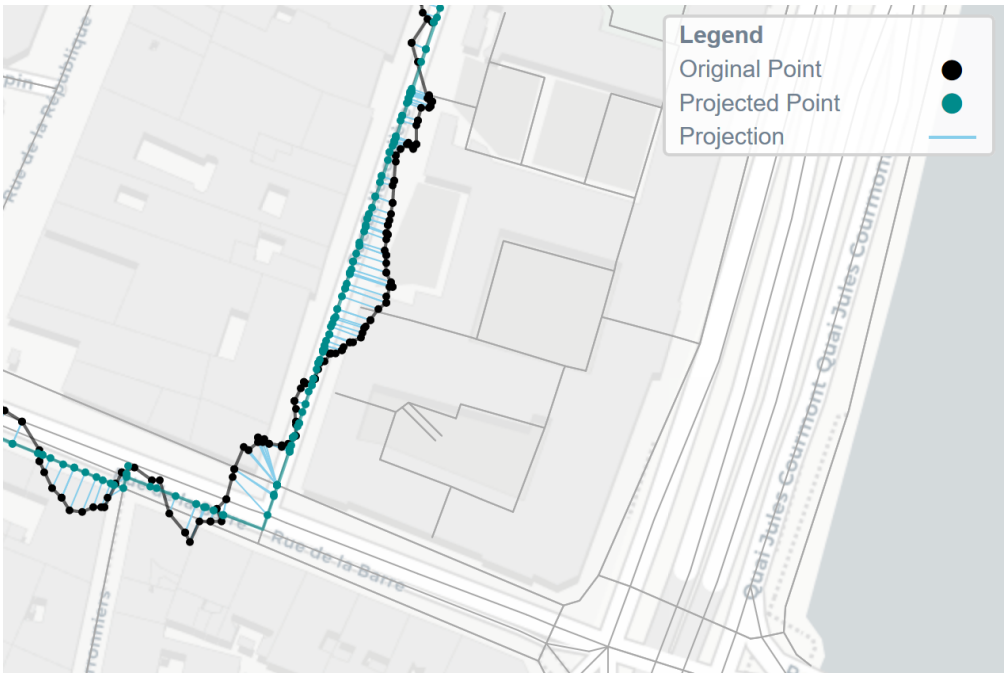
```
import numpy as np
import osmnx as ox
from noiseplanet import matcher
```

```
import numpy as np
import osmnx as ox
from noiseplanet import matching

track = np.array([[45.7584882 ,  4.83585996],
                  [45.75848068,  4.83586747],
                  [45.75849549,  4.83585205],
                  [45.75849134,  4.83584647],
                  [45.75848135,  4.8358245 ],
                  # ...
                  [45.75846756,  4.83580848],
                  [45.75844998,  4.83580936],
                  [45.7584067 ,  4.83580086],
                  [45.7584067 ,  4.83580086],
                  [45.75839346,  4.83579883]])

graph = matching.model.graph_from_track(track)

track_coor, route_corr, edgeid, stats = matching.match(graph, track, method='hmm')
```

And visualize the results :

from noiseplanet.ui import plot_html

```
# Plot the graph
plot_html(track, track_corr=track_corr, route_corr=route_corr,
        proj=True, show_graph=True)
```

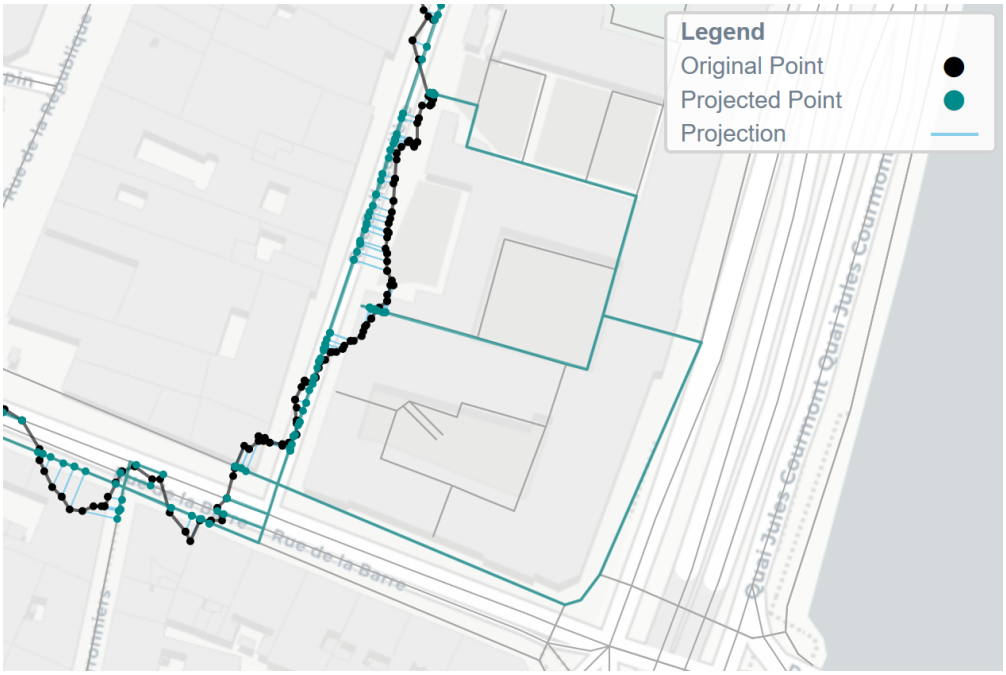



You can change the matching method. For example, the above method uses Hidden Markov Models, and is the best way to match a track on the OSM (see the report for more details). However, you can use a naive method to match the track on the nearest road :

```
track_coor, route_corr, edgeid, stats = matcher.match(graph, track, method='nearest')
```

And visualize the results :

```
from noiseplanet.ui import plot_html

# Plot the graph
plot_html(track, track_corr=track_corr, route_corr=route_corr,
        proj=True, show_graph=True)
```


Legend
Original Point
Projected Point
Projection
Rue de la Barre
Quai Jules Courmont

CHAPTER 4

# Packages

## 4.1 matcher

DataCleaner Module.

This module is under improvement. The aim is to clean and fix missing values in a pandas DataFrame.

noiseplanet.matcher.datacleaner.**clean_data**(*df*)
Clean a DataFrame track. This function is under development.

**Parameters** **df** (*pandas DataFrame*) – DataFrame to clean. Currently, only missing points are deleted.

**Returns** **df** – Cleaned DataFrame.

**Return type** pandas DataFrame

Matching Module.

Match a track/GeoJson to the Open Street Map network.

noiseplanet.matcher.matching.**match**(*graph*, *track*, *method='hmm'*)
Match a (Lat, Lon) track on the Open Street Map road network.

**Parameters**

- **graph** (*NetworkX MultiDiGraph*) – The OSM graph to match the track.
- **track** (*numpy 2D array*) – A 2D matrix composed by Latitudes (first column) and Longitudes (second column) of the track.
- **method** (*String, optional*) – Method used to match the track on the map. 'nearest' match the track on the nearest road. 'hmm' is a Hidden Markov Model based map matching algorithm. The default is 'hmm'.

**Returns**

- **track_corr** (*numpy 2D array*) – A 2D matrix composed by Latitudes (first column) and Longitudes (second column) of the track.

- **route_corr** (*numpy 2D array*) – A 2D matrix composed by Latitudes (first column) and Longitudes (second column) of the path connecting all track's points.
- **edgeid** (*numpy 2D array*) – List of edges to which each points belongs to. Edges id are composed by two extremity nodes id.
- **stats** (*Dict*) – Statistics of the Map Matching. 'proj_length' is the length of the projection (from track's point to corrected ones), 'path_length' is the distance on the graph between two following points, 'unlinked' higlights unconnected points on the graph.

noiseplanet.matcher.matching.**match_from_dir**(*dirname*, *out_dirname='.'*, *method='hmm'*, *log=True*)

Match a list of GeoJson tracks on the Open Street Map network.

**Parameters**

- **dirname** (*String*) – Path to the directory in which GeoJson tomatch are contained. All GeoJson files should contain track's points informations used to match on the network.
- **out_dirname** (*String, optional*) – Output directory path in which the corrected GeoJson tracks files are saved. The default is '.', the current working folder.
- **method** (*String, optional*) – Method used to match the track on the map. 'nearest' match the track on the nearest road. 'hmm' is a Hidden Markov Model based map matching algorithm. The default is 'hmm'.
- **log** (*Boolean, optional*) – Display console log of the Map Matching results. The default is True.

**Returns**

**Return type** None.

noiseplanet.matcher.matching.**match_from_geojson**(**file*, *out_dirname='.'*, *method='hmm'*, *log=True*)

Match a GeoJson track on the Open Street Map network.

**Parameters**

- **file** (*String*) – Path to the GeoJson file. It should contain track's points informations used to match on the network.
- **out_dirname** (*String, optional*) – Output directory path in which the corrected GeoJson track file is saved. The default is '.', the current working folder.
- **method** (*String, optional*) – Method used to match the track on the map. 'nearest' match the track on the nearest road. 'hmm' is a Hidden Markov Model based map matching algorithm. The default is 'hmm'.
- **log** (*Boolean, optional*) – Display console log of the Map Matching results. The default is True.

**Returns**

**Return type** None.

noiseplanet.matcher.matching.**match_geojson**(*geojson*, *method='hmm'*, *log=True*)

Match a GeoJson track on the Open Street Map network.

**Parameters**

- **geojson** (*Dict*) – GeoJson track to match on the Open Street Map network, on the dictionary format.

- **method** (*String, optional*) – Method used to match the track on the map. 'nearest' match the track on the nearest road. 'hmm' is a Hidden Markov Model based map matching algorithm. The default is 'hmm'.
- **log** (*Boolean, optional*) – Display console log of the Map Matching results. The default is True.

**Returns** **geojson_corr** – Corrected track on a GeoJson dictionary format.

**Return type** Dict

## 4.2 model

Route Module.

Track - OSM network interactions.

noiseplanet.matcher.model.route.**graph_from_track**(*track*, *network='all'*)

Get the OSM network for a given track.

**Parameters**

- **track** (*numpy 2D array*) – A 2D matrix composed by Latitudes (first column) and Longitudes (second column) of the track.
- **network** (*String*) – Network type extracted from the Open Street Map DataBase ('drive', 'pedestrian', 'all', etc. The default is 'all'.

**Returns** **graph** – Open Street Map graph of the track area.

**Return type** NetworkX MultiDiGraph

### Example

```
>>> import numpy as np
>>> from noiseplanet.matcher.model.route import graph_from_track
>>> track = np.array([[4.8396232, 45.7532804],
                      [4.839917548464699, 45.75345336404514],
                      [4.828226357067425, 45.7478253162003841])
>>> graph = graph_from_track(track, network='all')
>>> graph
    <networkx.classes.multidigraph.MultiDiGraph at 0x155de2260c8>
# Then you can visualize the graph, with matplotlib with :
>>> import osmnx as ox
>>> fig, ax = ox.plot_graph(graph)
# See the OSMNX documentation for more details
```

noiseplanet.matcher.model.route.**route_from_track**(*graph*, *track*, *edgeid=None*)

Get the route connecting points of a track projected on the Open Street Map network.

**Parameters**

- **graph** (*NetworkX MultiDiGraph*) – Open Street Map graph where the route will be computed.
- **track** (*numpy 2D array*) – A 2D matrix composed by Latitudes (first column) and Longitudes (second column) of the track.

- **edgeid** (*numpy 2D array, optional*) – List of edges to which each points belongs to. Edges id are composed by two extremity nodes id. The default is None.

**Returns**

- **route** (*numpy 2D array*) – Route connecting all track's points.
- **stats** (*Dict*) – Statistics of the routing connection. 'path_length' is the distance on the graph between two following points, 'unlinked' higlights unconnected points on the graph.

### Example

```
>>> import numpy as np
>>> from noiseplanet.matcher.model.route import route_from_track, graph_from_track
>>> track = np.array([[45.75809136,  4.83577159],
                      [45.7580932 ,  4.83576182],
                      [45.7580929 ,  4.8357634 ],
                      [45.75809207,  4.8357678 ],
                      [45.75809207,  4.8357678 ],
                      [45.75809647,  4.83574439],
                      [45.75809908,  4.83573054],
                      [45.75809908,  4.83573054],
                      [45.75810077,  4.83572153],
                      [45.75810182,  4.83571596]])
>>> graph = graph_from_track(track, network='all')
>>> route_corr, stats = route_from_track(graph, track)
```

Nearest Module.

Map Matching to the nearest edge.

noiseplanet.matcher.model.nearest.**match_nearest_edge**(*graph*, *track*)

Algorithm to match the track to the nearest edge on the Open Street Map network.

This function match a track of GPS coordinates, in the (Lat, Lon) format. to a graph.

It loops all over the points to match them to the closest edge of the OSM network. The GPS points are projected on the edge with an orthogonal projection. If the projected point goes outside of the edge, then it is match to one extremity (see noiseplanet.utils.oproj documentation for more details).

**Parameters**

- **graph** (*NetworkX MultiDiGraph*) – Graph of the Open Street Map network.
- **track** (*numpy 2D array*) – A 2D matrix composed by Latitudes (first column) and Longitudes (second column) of the track.

**Returns**

- **track_corr** (*numpy 2D array*) – A 2D matrix composed by Latitudes (first column) and Longitudes (second column) of the corrected track.
- **route** (*numpy 2D array*) – Route connecting all track's points on the Open Street Map network.
- **edgeid** (*numpy 2D array*) – List of edges to which each points belongs to. Edges id are composed by two extremity nodes id.
- **stats** (*Dict*) – Statistics of the Map Matching. 'proj_length' is the length of the projection (from track's point to corrected ones), 'path_length' is the distance on the graph between two following points, 'unlinked' higlights unconnected points on the graph.

### Example

```
>>> import osmnx as ox
>>> import numpy as np
>>> from noiseplanet.matcher.model.leuven import leuven
>>> place_name = "2e Arrondissement, Lyon, France"
>>> distance = 1000  # meters
>>> graph = ox.graph_from_address(place_name, distance)
>>> track = np.array([[45.75809136,  4.83577159],
                      [45.7580932 ,  4.83576182],
                      [45.7580929 ,  4.8357634 ],
                      [45.75809207,  4.8357678 ],
                      [45.75809207,  4.8357678 ],
                      [45.75809647,  4.83574439],
                      [45.75809908,  4.83573054],
                      [45.75809908,  4.83573054],
                      [45.75810077,  4.83572153],
                      [45.75810182,  4.83571596],
                      [45.75810159,  4.83571719],
                      [45.7581021 ,  4.83571442],
                      [45.7580448 ,  4.83558152],
                      [45.75804304,  4.83558066],
                      [45.75804304,  4.83558066],
                      [45.75802703,  4.83557288]])
>>> track_corr, route_corr, edgeid = match_nearest(graph, track)
```

Leuven Module. Map Matching from Leuven algorithm.

noiseplanet.matcher.model.leuven.**match_leuven**(*graph*, *track*)

Algorithm to match the track to the most probable route.

Rely on Leuven Map Matching package. Copyright 2015-2018, KU Leuven - DTAI Research Group, Sirris - Elucidata Group. See the docs : https://leuvenmapmatching.readthedocs.io/en/latest/index.html

If you want custom Distance Matcher, then change 'matcher' on this function.

**Parameters**

- **graph** (*NetworkX MultiDiGraph*) – Graph of the Open Street Map network.
- **track** (*numpy 2D array*) – A 2D matrix composed by Latitudes (first column) and Longitudes (second column) of the track.

**Returns**

- **track_corr** (*numpy 2D array*) – A 2D matrix composed by Latitudes (first column) and Longitudes (second column) of the corrected track.
- **route** (*numpy 2D array*) – A 2D matrix composed by Latitudes (first column) and Longitudes (second column) of the path connecting all track's points.
- **edgeid** (*numpy 2D array*) – List of edges to which each points belongs to. Edges id are composed by two extremity nodes id.
- **stats** (*Dict*) – Statistics of the Map Matching. 'proj_length' is the length of the projection (from track's point to corrected ones), 'path_length' is the distance on the graph between two following points, 'unlinked' higlights unconnected points on the graph.

### Example

```
>>> import osmnx as ox
>>> import numpy as np
>>> from noiseplanet.matcher.model.leuven import match_leuven
>>> place_name = "2e Arrondissement, Lyon, France"
>>> distance = 1000  # meters
>>> graph = ox.graph_from_address(place_name, distance)
>>> track = np.array([[45.75809136,  4.83577159],
                      [45.7580932 ,  4.83576182],
                      [45.7580929 ,  4.8357634 ],
                      [45.75809207,  4.8357678 ],
                      [45.75809207,  4.8357678 ],
                      [45.75809647,  4.83574439],
                      [45.75809908,  4.83573054],
                      [45.75809908,  4.83573054],
                      [45.75810077,  4.83572153],
                      [45.75810182,  4.83571596],
                      [45.75810159,  4.83571719],
                      [45.7581021 ,  4.83571442],
                      [45.7580448 ,  4.83558152],
                      [45.75804304,  4.83558066],
                      [45.75804304,  4.83558066],
                      [45.75802703,  4.83557288]])
>>> track_corr, route_corr, edgeid = match_leuven(graph, track)
```

## 4.3 utils

Functions Module.

This module convert geojson to pandas DataFrame.

noiseplanet.utils.functions.**df_to_geojson**(*df*, *geometry*, *coordinates*, *properties*)

Convert a pandas DataFrame to a GeoJson dictionary.

**Parameters**

- **df** (*pandas DataFrame*) – DataFrame containing the geojson's informations.
- **geometry** (*String*) – The type of the geometry (Point, Polygon, etc.).
- **coordinates** (*String*) – The DataFrame column's name of the coordinates.
- **properties** (*list of String eelements.*) – The DataFrame column's names of the properties attributes.

**Returns** **geojson** – GeoJson dictionary with the geometry, coordinates, and properties elements.

**Return type** Dict

noiseplanet.utils.functions.**geojson_to_df**(*geojson*, *normalize_header=False*)

Convert a GeoJson dictionary to a pandas DataFrame.

**Parameters**

- **geojson** (*Dict*) – GeoJson dictionary.
- **normalize_header** (*Boolean, optional*) – If True, normalize the header by spliting the path name and only keeping the attribute name. The default is False.

**Returns** **df** – Converted GeoJson in a DataFrame format.

**Return type** pandas DataFrame

### Example

```
>>> from noiseplanet.utils.functions import geojson_to_df
>>> df = geojson_to_df(geojson, normalize_header=False)
>>> df.head
    type      geometry.type   geometry.coordinates    properties.id ...
    Feature   Point           [4.52, 45.58, 201.15]   0
    ...
```

```
>>> df = geojson_to_df(geojson, normalize_header=True)
>>> df.head
    type            coordinates             id ...
    Point           [4.52, 45.58, 201.15]   0
    ...
```

OProj Module.

Compute orthogonal projection on lines and segment.

noiseplanet.utils.oproj.**distance_great_circle**(*pointA*, *pointB*)
Compute the Great Circle Distance between two points.

**Parameters**

- **pointA** (*Tuple*) – Pooint A.
- **pointB** (*Tuple*) – Point B.

**Returns** **distance** – Distance between A and B.

**Return type** Float

noiseplanet.utils.oproj.**distance_haversine**(*pointA*, *pointB*)
Compute the Haversine distance between two points, in Lat Lon.

**Parameters**

- **pointA** (*Tuple*) – Point A in (Lat, Lon) format.
- **pointB** (*Tuple*) – Point B in (Lat, Lon) format.

**Returns** **distance** – Distance between A and B.

**Return type** Float

noiseplanet.utils.oproj.**orthoProj**(*pointA*, *pointB*, *S*)
Orthogonal projection of a point on a line.

**Parameters**

- **pointA** (*Tuple*) – Point to project on the line.
- **pointB** (*Tuple*) – Point belonging to the line.
- **S** (*Tuple*) – Slope of the line.

**Returns** **projection** – Porjection of Point A on the line (Point B, Slope).

**Return type** Tuple

noiseplanet.utils.oproj.**orthoProjSegment**(*pointA*, *pointB*, *pointC*)

Orthogonal projection of a point into a line.

**Parameters**

- **pointA** (*Tuple*) – Point to project on the line.
- **pointB** (*Tuple*) – Point belonging to the line.
- **pointC** (*Tuple*) – Second point belonging to the line.

**Returns** **projection** – Porjection of Point A on the line (Point B, Slope).

**Return type** Tuple

noiseplanet.utils.oproj.**slope**(*pointA*, *pointB*)

Compute the slope (directional coefficient) of a line from two points.

**Parameters**

- **pointA** (*Tuple*) – Pooint A.
- **pointB** (*Tuple*) – Point B.

**Returns** **slope** – Slope vector from point A to point B.

**Return type** Tuple

HexGrid Module.

Generate Hexagonal grid. Cartesian-Hexagonal coordinates interaction.

noiseplanet.utils.hexgrid.**cartesian_to_hex**(*point*, *origin=(0, 0)*, *side_length=1*, *proj_init=None*, *proj_out=None*)

Convert cartesian coordinates to hexagonal coordinates system.

**Parameters**

- **point** (*Tuple*) – Point in cartesian coordinates to convert.
- **origin** (*Tuple, optional*) – Origin of the hexagonal coordinates system. The default is (0, 0).
- **side_length** (*Float, optional*) – Side length of the hexagons. The default is 1.
- **proj_init** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_init refers to the starting coordinates system. The default is None.
- **proj_out** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_out refers to the ending coordinates system. The default is None.

  **Example :** If the bbox is in geographic coordinates, but the hexgrid should be computed on the web mercator system. Then, >>> proj_init="epsg:4326" >>> proj_out="epsg:3857"

**Returns** **hex_coord** – Point's coordinates in hexagonal coordinates system.

**Return type** Tuple

noiseplanet.utils.hexgrid.**cartesians_to_hexs**(*X*, *Y*, *origin=(0, 0)*, *side_length=1*, *proj_init=None*, *proj_out=None*)

Convert a list of cartesian points to hexagonal coordinates.

**Parameters**

- **X** (*numpy 1D array*) – X indexes of cartesian points.

- **Y** (*numpy 1D array*) – Y indexes of cartesian points.
- **origin** (*Tuple, optional*) – Origin of the hexagonal coordinates system. The default is (0, 0).
- **side_length** (*Float, optional*) – Side length of the hexagons. The default is 1.
- **proj_init** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_init refers to the starting coordinates system. The default is None.
- **proj_out** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_out refers to the ending coordinates system. The default is None.

  **Example :** If the bbox is in geographic coordinates, but the hexgrid should be computed on the web mercator system. Then, >>> proj_init="epsg:4326" >>> proj_out="epsg:3857"

**Returns** **hexagons** – Cartesian points in hexagonal coordinates.

**Return type** numpy 2D array

noiseplanet.utils.hexgrid.**hex_to_cartesian**(*hexa*, *origin=(0, 0)*, *side_length=1*, *proj_init=None*, *proj_out=None*)

Convert hexagonal coordinates to cartesian.

**Parameters**

- **hexa** (*Tuple*) – Hexagonal coordinates.
- **origin** (*Tuple, optional*) – Origin of the hexagonal coordinates system. The default is (0, 0).
- **side_length** (*Float, optional*) – Side length of the hexagons. The default is 1.
- **proj_init** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_init refers to the starting coordinates system. The default is None.
- **proj_out** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_out refers to the ending coordinates system. The default is None.

  **Example :** If the bbox is in geographic coordinates, but the hexgrid should be computed on the web mercator system. Then, >>> proj_init="epsg:4326" >>> proj_out="epsg:3857"

**Returns** **point** – Hexagon's coordinates in cartesian.

**Return type** Tuple

noiseplanet.utils.hexgrid.**hexagon_coordinates**(*center*, *side_length=1*, *r=0.8660254037844389*, *R=1.0000000000000002*, *proj_init=None*, *proj_out=None*)

Get the hexagon's coordinates points from its center.

**Parameters**

- **center** (*Tuple*) – Center of the hexagon, in cartesian coordinates.
- **side_length** (*Float, optional*) – Side length of the hexagon. The default is 1.
- **r** (*Float, optional*) – Intern radius. The default is 0.8660254037844389.
- **R** (*Float, optional*) – Extern radius. The default is 1.0000000000000002.

- **proj_init** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_init refers to the starting coordinates system. The default is None.
- **proj_out** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_out refers to the ending coordinates system. The default is None.

  **Example :** If the bbox is in geographic coordinates, but the hexgrid should be computed on the web mercator system. Then, >>> proj_init="epsg:4326" >>> proj_out="epsg:3857"

**Returns** **hexagon** – List of points belonging to the hexagon.

**Return type** List

noiseplanet.utils.hexgrid.**hexagons_coordinates**(*X*, *Y*, *side_length=1*, *r=0.8660254037844389*, *R=1.0000000000000002*, *proj_init=None*, *proj_out=None*)

Get the hexagons' coordinates points from a list of center.

**Parameters**

- **X** (*numpy 1D array*) – X indexes of cartesian center.
- **Y** (*numpy 1D array*) – Y indexes of cartesian center.
- **side_length** (*Float, optional*) – Side length of the hexagon. The default is 1.
- **r** (*Float, optional*) – Intern radius. The default is 0.8660254037844389.
- **R** (*Float, optional*) – Extern radius. The default is 1.0000000000000002.
- **proj_init** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_init refers to the starting coordinates system. The default is None.
- **proj_out** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_out refers to the ending coordinates system. The default is None.

  **Example :** If the bbox is in geographic coordinates, but the hexgrid should be computed on the web mercator system. Then, >>> proj_init="epsg:4326" >>> proj_out="epsg:3857"

**Returns** **hexagons** – List of hexagons, composed by their coordinates.

**Return type** List

noiseplanet.utils.hexgrid.**hexbin_grid**(*bbox*, *side_length=1*, *proj_init=None*, *proj_out=None*)

Create a grid of hexagons.

See http://www.calculatorsoup.com/calculators/geometry-plane/polygon.php

**Parameters**

- **bbox** (*Tuple*) – Box of the area to generate the hexagons. Format : Lower X, Lower Y, Upper X, Upper Y.
- **side_length** (*float, optional*) – Side length of the hexagons. The default is 1.
- **proj_init** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_init refers to the starting coordinates system. The default is None.

- **proj_out** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_out refers to the ending coordinates system. The default is None.

### Example

If the bbox is in geographic coordinates, but the hexgrid should be computed on the web mercator system. Then, >>> proj_init="epsg:4326" >>> proj_out="epsg:3857"

**Returns** **polygons** – List of hexagons. An hexagons is a list of coordinates (tuple, Lat, Lon).

**Return type** List

noiseplanet.utils.hexgrid.**hexs_to_cartesians**(*Q*, *R*, *origin=(0, 0)*, *side_length=1*, *proj_init=None*, *proj_out=None*)

Convert a list of hexagonal coordinates to cartesian.

**Parameters**

- **Q** (*numpy 1D array*) – Columns indexes of hexagons coordinates.
- **R** (*numpy 1D array*) – Rows indexes of hexagons coordinates.
- **origin** (*Tuple, optional*) – Origin of the hexagonal coordinates system. The default is (0, 0).
- **side_length** (*Float, optional*) – Side length of the hexagons. The default is 1.
- **proj_init** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_init refers to the starting coordinates system. The default is None.
- **proj_out** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_out refers to the ending coordinates system. The default is None.

  **Example :** If the bbox is in geographic coordinates, but the hexgrid should be computed on the web mercator system. Then, >>> proj_init="epsg:4326" >>> proj_out="epsg:3857"

**Returns** **points** – Hexagons points in cartesian coordinates.

**Return type** numpy 2D array

noiseplanet.utils.hexgrid.**nearest_hexagon**(*point*, *origin=(0, 0)*, *side_length=1*, *proj_init=None*, *proj_out=None*)

Get the nearest hexagon center from a cartesian point.

**Parameters**

- **point** (*Tuple*) – Cartesian point.
- **origin** (*Tuple, optional*) – Origin of the hexagonal coordinates system. The default is (0, 0).
- **side_length** (*Float, optional*) – Side length of the hexagons. The default is 1.
- **proj_init** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_init refers to the starting coordinates system. The default is None.
- **proj_out** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_out refers to the ending coordinates system. The default is None.

**Example :** If the bbox is in geographic coordinates, but the hexgrid should be computed on the web mercator system. Then, >>> proj_init="epsg:4326" >>> proj_out="epsg:3857"

**Returns** **hexagon_center** – Hexagonal coordinates of the nearest hexagon from point.

**Return type** Tuple

noiseplanet.utils.hexgrid.**nearest_hexagons**(*X*, *Y*, *origin=(0, 0)*, *side_length=1*, *proj_init=None*, *proj_out=None*)

Get the nearest hexagons centers from a list of cartesian points.

**Parameters**

- **X** (*numpy 1D array*) – X indexes of cartesian points.
- **Y** (*numpy 1D array*) – Y indexes of cartesian points.
- **origin** (*Tuple, optional*) – Origin of the hexagonal coordinates system. The default is (0, 0).
- **side_length** (*Float, optional*) – Side length of the hexagons. The default is 1.
- **proj_init** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_init refers to the starting coordinates system. The default is None.
- **proj_out** (*String, optional*) – If working with coordinates and the hexagons need to be calculated in another coordinates system, proj_out refers to the ending coordinates system. The default is None.

  **Example :** If the bbox is in geographic coordinates, but the hexgrid should be computed on the web mercator system. Then, >>> proj_init="epsg:4326" >>> proj_out="epsg:3857"

**Returns** **hexagons_center** – Cartesian points in hexagonal coordinates.

**Return type** numpy 2D array

## 4.4 io

InputOutput Module.

This module save and open files/directory.

noiseplanet.io.inputoutput.**extract_track**(*query_csv*, *out_dir='data'*)

Extract GeoJson tracks from a CSV.

See https://dashboard.noise-planet.org/public/question/52ee2bde-2d28-4377-bcbb-061d7cbfa343

**Parameters** **query_csv** (*String*) – CSV file of the tracks.

**Returns**

**Return type** None.

noiseplanet.io.inputoutput.**open_files**(*dir_path*, *ext='geojson'*)

Get the path of all files in a directory

**Parameters**

- **dir_path** (*string*) – Name or path to the directory where the files you want to open are saved..
- **ext** (*string, optional*) – The files extension you want to open. The default is "geojson".

**Raises** `FileNotFoundError` – If the directory is not found, it raises this issue.

**Returns** A list of the path of all files saved in the directory, with the extension geojson by default.

**Return type** array like

### Example

```
>>> dir_name = "path/to/your/directory"
>>> files = open_files(dir_name, ext="geojson")
>>> files
    ['../test/track_test/track.geojson',
     '../test/track_test/track(1).geojson',
     '../test/track_test/track(2).geojson',
     ...
     '../test/track_test/track(100).geojson']
```

`noiseplanet.io.inputoutput.`**`open_geojson`**(*file_path*)

Open a GeoJson file in a dictionary format.

**Parameters** **`file_path`** (*`String`*) – Path / Name of the Geojson file. Should contains the extension.

**Returns** **geojson** – Dictionary of the GeoJson.

**Return type** Dict

`noiseplanet.io.inputoutput.`**`open_properties`**(*file_path*, *sep='='*, *comment_char='#'*)

Open properties file.

**Parameters**

- **`file_path`** (*`String`*) – Path / Name of the properties file.
- **`sep`** (*`String, optional`*) – Separator of attributes / elements. The default is '='.
- **`comment_char`** (*`String, optional`*) – Comment symbol. The default is '#'.

**Returns** **properties** – Dictionary of the properties.

**Return type** Dict

`noiseplanet.io.inputoutput.`**`save_geojson`**(*geojson*, *out_path*)

Save a GeoJson dictionary.

**Parameters**

- **`geojson`** (*`Dict`*) – Dictionary of the GeoJson.
- **`out_path`** (*`String`*) – Output path / name of the Geojson file. Should contains the extension.

**Returns**

**Return type** None.

`noiseplanet.io.inputoutput.`**`unzip_dir`**(*in_dir*, *out_dir*)

Unzip files in a directory.

**Parameters**

- **`out_dir`** (*`String`*) – Path to the output location.
- **`*file`** (*`String`*) – Path of files to unzip.

**Returns**

**Return type** None.

noiseplanet.io.inputoutput.**unzip_file**(**file*, *out_dir*)
Unzip files in a directory.

**Parameters**

- **out_dir** (*String*) – Path to the output location.
- ***file** (*String*) – Path of files to unzip.

**Returns**

**Return type** None.

## 4.5 db

DBConnect Module.

Connection to DataBase.

noiseplanet.db.connect.**connect**(*db_file*)
Create a DataBase connection to a SQLite DataBase.

**Parameters** **db_file** (*String*) – Path to the DataBase file. If the DataBase does not exists, a new one is created.

**Returns** **conn** – Connection to the DataBase db_file.

**Return type** SQLite3 Connection

noiseplanet.db.connect.**database_query**(*conn*, *query*)
Request a query in a database.

**Parameters**

- **conn** (*SQLite3 Connection*) – Connection to the DataBase where the query is requested.
- **create_table_sql** (*String*) – SQL query to create a table in.

**Returns**

**Return type** None.

DBConnect Module. Add DataFrame/Values to a DataBase.

noiseplanet.db.commit.**df_to_table**(*conn*, *table_name*, *df*)
Paste Values from a DataFrame into a DataBase table.

**Parameters**

- **conn** (*SQLite3 Connection*) – The connection to the DataBase.
- **table_name** (*String*) – Name of the DataBase's table where the DataFrame will be pasted.
- **df** (*pandas DataFrame*) – DataFrame to transfer into the DataBase.

**Returns**

**Return type** None.

noiseplanet.db.commit.**geojson_to_table**(*conn*, *table_name*, **file_geojson*)
Add a GeoJson file to the table in a SQLite DataBase.

**Parameters**

- **conn** (*SQLite3 Connection*) – The connection to the DataBase.
- **table_name** (*String*) – Name of the DataBase's table where the DataFrame will be pasted.
- ***file_geojson** (*String*) – GeoJson files to add.

**Returns**

**Return type** None.

noiseplanet.db.commit.**properties_to_table**(*conn*, *table_name*, **file_properties*)
Add a properties file to the table in a SQLite DataBase.

**Parameters**

- **conn** (*SQLite3 Connection*) – The connection to the DataBase.
- **table_name** (*String*) – Name of the DataBase's table where the DataFrame will be pasted.
- ***file_properties** (*String*) – Properties files to add.

**Returns**

**Return type** None.

noiseplanet.db.commit.**select_to_df**(*conn*, *query*)
Convert the query request into a DataFrame.

**Parameters**

- **conn** (*SQLite3 Connection*) – Connection to the DataBase.
- **query** (*String*) – Query to request on the DataBase.

**Returns** **df** – The result of the Query, in a DataFrame format.

**Return type** pandas DataFrame

noiseplanet.db.commit.**track_to_db**(*conn*, *dir_track*)
Add tracks informations in a SQLite DataBase.

**Parameters**

- **conn** (*SQLite3 Connection*) – The connection to the DataBase.
- **dir_track** (*String*) – Path to the track files.

**Returns**

**Return type** None.

CHAPTER 5

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## n

# Index

## O



## P



## R



## S



## T



## U