Track Fixity Layer - Final Report

Technical Documentation

Version 0.0.1

Qian Fu

John M. Easton

Michael P. N. Burrow

School of Engineering, University of Birmingham

First created: **August 2020**

Last updated: **July 2024**

Table of Contents

List of Figures

Chapter 1

Introduction

The "Track Fixity Layer" project aims to create a data mining tool to help track engineers measure, predict and analyse track fixity parameters for any railway section. These parameters include the rate and direction of rail head movement in both horizontal and vertical planes relative to the rail plane. A prototype machine learning model has been developed to predict these parameters by integrating diverse data sources related to factors influencing track fixity. The methodology has been tested and validated with real data from a selected section of the East Coast Main Line railway.

The project has resulted in the following key achievements:

- **Data integration and management**: A data cleaning and pre-processing workflow has been developed using the open-source PostgreSQL database. This workflow enables smooth integration and management of the data corpus. Initially, six industry data streams were used (CARRS, CNM, OPAS, ballast type, track quality, and point cloud surveys). The workflow is easily extensible, allowing for the inclusion of additional data sets in the future, provided they are referenced linearly (e.g., ELR + track mileage) or using standard geographic coordinate systems.
- **Predictive machine learning model**: A machine learning model was trained on data from an 80 km section of the ECM8 line south of Edinburgh. This model predicts future track movements, with most predictions falling within the correct bin or within a single bin width of the true value. The model primarily used parameters such as curvature, cant, and maximum speed for its predictions.
- **Python package development**: A three-module Python package has been developed to facilitate the rapid implementation of the described functionalities in software workflows.

These achievements lay the groundwork for further development and refinement in future phases of the project.

This document provides a detailed description of the developed data mining tool's functionality. The source code, written in Python, is available for download at the following GitHub repository.

1.1 Methodology

The methodology for this project is illustrated in Figure 1.

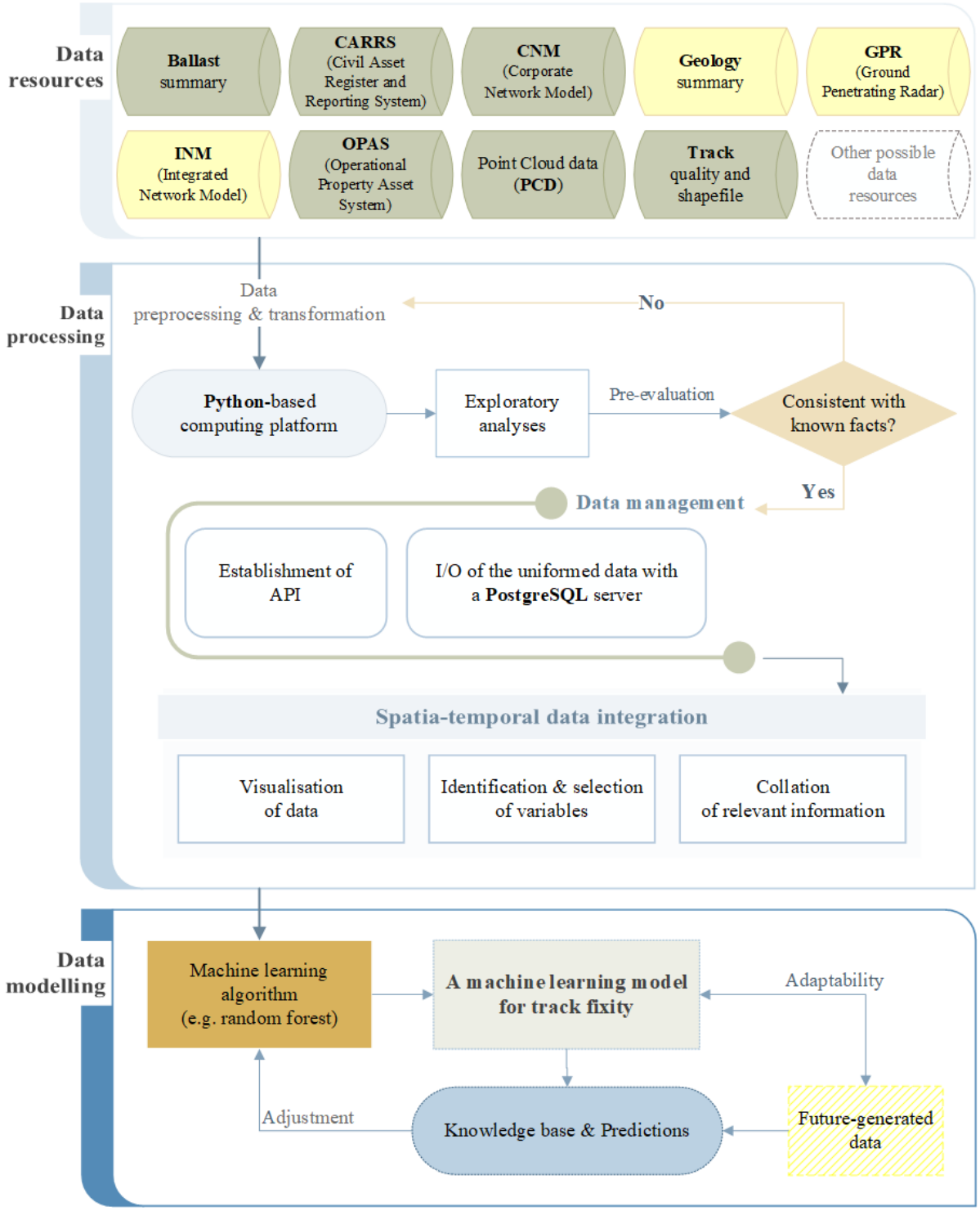


Figure 1: The methodological framework designed and implemented for the project.

1.2 Database

All available data resources, which are preprocessed by the module *preprocessor*, are stored in different schemas of a PostgreSQL database - "NR_TrackFixity". A snapshot for the schemas in the database is illustrated in Figure 2.

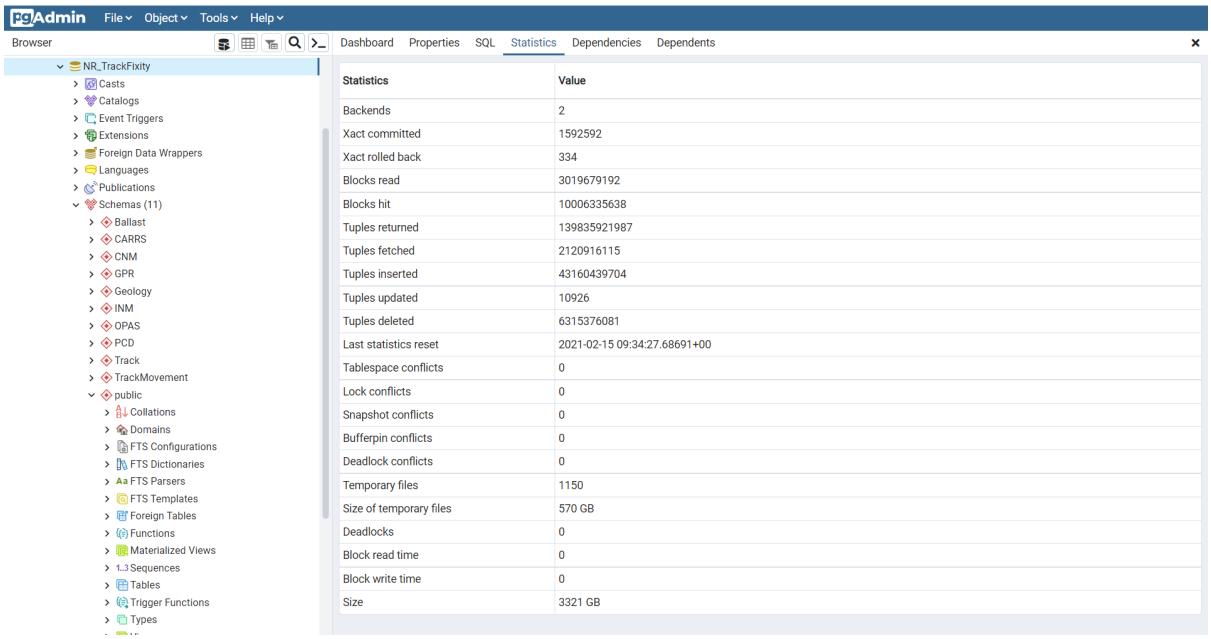


Figure 2: Snapshot of the schemas in the project database "NR_TrackFixity".

Chapter 2

Modules

utils	Helper classes/functions for facilitating the implementation of other modules.
preprocessor	Preprocessing and I/O management of all the data resources that are made available for the project.
shaft	Further processing and exploration of the data that are preprocessed by the subpackage *preprocessor* to generate a comprehensive data set for the development of a machine learning model on track fixity.
modeller	Applying machine learning algorithms to data generated from *shaft* to create a model for track movement prediction.

2.1 utils

Helper classes/functions for facilitating the implementation of other modules.

2.1.1 General utility tools

Classes:

TrackFixityDB([host, port, username, ...])	A class inheriting from pyhelpers.dbms.PostgreSQL as a basic instance of PostgreSQL for managing data of the project.

TrackFixityDB

class src.utils.**TrackFixityDB**(*host=None*, *port=None*, *username=None*, *password=None*, *database_name='NR_TrackFixity'*, ***kwargs*)

A class inheriting from pyhelpers.dbms.PostgreSQL as a basic instance of PostgreSQL for managing data of the project.

Parameters

- **host** (*str | None*) – The database host; defaults to None.
- **port** (*int | None*) – The database port; defaults to None.
- **username** (*str | None*) – The database username; defaults to None.
- **password** (*str | int | None*) – The database password; defaults to None.
- **database_name** (*str*) – The name of the database; defaults to NR_TrackFixity.
- **kwargs** – [Optional] parameters of the class pyhelpers.dbms.PostgreSQL.

Examples:

```
>>> from src.utils import TrackFixityDB
>>> db_instance = TrackFixityDB(host='localhost')  # Connect the default local server
Password (postgres@localhost:5432): ***
Connecting postgres:***@localhost:5432/NR_TrackFixity ... Successfully.
>>> db_instance.database_name
'NR_TrackFixity'
>>> db_instance = TrackFixityDB()  # Remote server
>>> db_instance.database_name
'NR_TrackFixity'
```

Note: No directly defined attributes. See inherited class attributes.

Note: No directly defined methods. See inherited class methods.

Functions:

cd_docs_source(*sub_dir[, mkdir])	Get path to `data_dir` and/or subdirectories / files.
eval_data_type(str_val)	Convert a string to its intrinsic data type.
year_month_to_data_date(year, month[, fmt])	Convert a pair of `year` and `month` to a formatted date.
data_date_to_year_month(data_date[, fmt])	Convert a formatted date to a pair of `year` and `month`.
add_sql_query_date_condition(sql_query, ...)	Add a condition about data date to a given SQL query statement.
get_tile_xy(tile_xy)	Get X and Y coordinates of a tile.
add_sql_query_xy_condition(sql_query[, tile_xy])	Add a condition about the tiles of point cloud data to a given SQL query statement.
iterable_to_range(iterable)	Convert an iterable of sequential integers to a range or ranges.
paired_next(iterable)	Loop through an iterable for every two neighbouring elements.
grouped(iterable, n)	Group every two elements of an iterable variable.
add_sql_query_elr_condition(sql_query[, elr])	Add a condition about specific ELR(s) to a given SQL query statement.
numba_np_shift(array, step[, fill_value])	Shift an array by desired number of rows with Numba.
find_valid_names(name, valid_names)	Find valid names.
fix_folium_float_image(path_to_m)	Fix incorrect display of floating images (when utilising folium<=0.13.0).

src.utils.cd_docs_source

src.utils.**cd_docs_source**(**sub_dir*, *mkdir=False*, ***kwargs*)

Get path to `data_dir` and/or subdirectories / files.

Parameters

- **sub_dir** (*str*) – Name of directory or names of directories (and/or a filename).
- **mkdir** (*bool*) – Whether to create a directory; defaults to `False`.
- **kwargs** – [Optional] parameters of os.makedirs, e.g. `mode=0o777`.

Return path

An absolute path to a directory (or a file) under `data_dir`.

Return type

str

Examples:

```
>>> from src.utils import cd_docs_source
>>> import os
>>> path_to_docs_source = cd_docs_source()
>>> os.path.relpath(path_to_docs_source)
'docs\source'
>>> path_to_docs_img = cd_docs_source("_images")
>>> os.path.relpath(path_to_docs_img)
'docs\source\_images'
```

src.utils.eval_data_type

src.utils.**eval_data_type**(*str_val*)

Convert a string to its intrinsic data type.

Parameters

str_val (*str*) – A string-type variable.

Returns

Converted value.

Return type

Any

Examples:

```
>>> from src.utils import eval_data_type
>>> val_1 = '1'
>>> origin_val = eval_data_type(val_1)
>>> type(origin_val)
int
>>> origin_val
1
>>> val_2 = '1.1.1'
>>> origin_val = eval_data_type(val_2)
>>> type(origin_val)
str
>>> origin_val
'1.1.1'
```

src.utils.year_month_to_data_date

src.utils.**year_month_to_data_date**(*year*, *month*, *fmt='%Y%m'*)

Convert a pair of `year` and `month` to a formatted date.

Parameters

- **year** (*str | int*) – Year of when data was collected.
- **month** (*str | int*) – Month of when data was collected.
- **fmt** (*str*) – Format; defaults to `'%Y%m'`.

Returns

Formatted date of the data.

Return type
str

Examples:

```
>>> from src.utils import year_month_to_data_date
>>> year_month_to_data_date(year=2019, month=10)
'201910'
>>> year_month_to_data_date(year=2020, month=4)
'202004'
```

src.utils.data_date_to_year_month

src.utils.**data_date_to_year_month**(*data_date*, *fmt='%Y%m'*)

Convert a formatted date to a pair of `year` and `month`.

Parameters

- **data_date** (*str | datetime.datetime | datetime.date | int*) – Date of when the data was collected.
- **fmt** (*str*) – Format of the date string; defaults to `'%Y%m'`.

Returns
Year and month of when data was collected.

Return type
tuple

Examples:

```
>>> from src.utils import data_date_to_year_month
>>> data_date_to_year_month(data_date='201910')
(2019, 10)
>>> data_date_to_year_month(data_date=10, fmt='%m')
(None, 10)
>>> data_date_to_year_month(data_date=20, fmt='%y')
(2020, None)
```

src.utils.add_sql_query_date_condition

src.utils.**add_sql_query_date_condition**(*sql_query*, *data_date*, *date_fmt='%Y%m'*)

Add a condition about data date to a given SQL query statement.

Parameters

- **sql_query** (*str*) – SQL query statement.
- **data_date** (*str | datetime.datetime | datetime.date | int | list | tuple*) – Date of data.
- **date_fmt** (*str*) – Format of the date string; defaults to `'%Y%m'`.

Returns
Updated SQL query statement with conditions about data date.

Return type
str

Examples:

```
>>> from src.utils import add_sql_query_date_condition
>>> query1 = 'SELECT * FROM a_table'
>>> query1
'SELECT * FROM a_table'
>>> add_sql_query_date_condition(query1, data_date=201910)
'SELECT * FROM a_table WHERE "Year"=2019 AND "Month"=10'
>>> add_sql_query_date_condition(query1, data_date='202004')
'SELECT * FROM a_table WHERE "Year"=2020 AND "Month"=4'
>>> query2 = 'SELECT * FROM X where a=b'
>>> query2
'SELECT * FROM X where a=b'
>>> add_sql_query_date_condition(query2, data_date=['201910', '202004'])
'SELECT * FROM X where a=b AND "Year" IN (2019, 2020) AND "Month" IN (10, 4)'
```

src.utils.get_tile_xy

src.utils.**get_tile_xy**(*tile_xy*)

Get X and Y coordinates of a tile.

Parameters
tile_xy (*tuple | list | str*) – X and Y coordinates in reference to a tile for the point cloud data.

Returns
X and Y coordinates of a tile.

Return type
tuple

Examples:

```
>>> from src.utils import get_tile_xy
>>> tile_x, tile_y = get_tile_xy(tile_xy="Tile_X+0000340500_Y+0000674200.laz")
>>> tile_x, tile_y
(340500, 674200)
>>> tile_x, tile_y = get_tile_xy(tile_xy=(340500, 674200))
>>> tile_x, tile_y
(340500, 674200)
>>> tile_x, tile_y = get_tile_xy(tile_xy='X340500_Y674200')
>>> tile_x, tile_y
(340500, 674200)
>>> tile_x, tile_y = get_tile_xy(tile_xy=(340500, None))
>>> # tile_x, tile_y = get_tile_xy(tile_xy=(340500, ''))
>>> tile_x, tile_y
(340500, None)
```

src.utils.add_sql_query_xy_condition

src.utils.**add_sql_query_xy_condition**(*sql_query*, *tile_xy=None*)

Add a condition about the tiles of point cloud data to a given SQL query statement.

Parameters

- **sql_query** (*str*) – SQL query statement.
- **tile_xy** (*tuple | list | str | None*) – Easting (X) and northing (Y) of the geographic Cartesian coordinates for a tile; defaults to None.

Returns

Updated SQL query statement with the tiles of point cloud data.

Return type

str

Examples:

```
>>> from src.utils import add_sql_query_xy_condition
>>> query = 'SELECT * FROM a_table'
>>> query
'SELECT * FROM a_table'
>>> add_sql_query_xy_condition(query, tile_xy=(340500, 674200))
'SELECT * FROM a_table WHERE "Tile_X"=340500 AND "Tile_Y"=674200'
>>> add_sql_query_xy_condition(query, tile_xy=(340500, None))
'SELECT * FROM a_table WHERE "Tile_X"=340500'
```

src.utils.iterable_to_range

src.utils.**iterable_to_range**(*iterable*)

Convert an iterable of sequential integers to a range or ranges.

Parameters

iterable (*Iterable*) – An iterable of sequential integers.

Returns

A range or ranges.

Return type

generator

Examples:

```
>>> from src.utils import iterable_to_range
>>> lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> rng = iterable_to_range(lst)
>>> type(rng)
generator
>>> list(rng)
[(1, 9)]
```

src.utils.paired_next

src.utils.**paired_next**(*iterable*)

Loop through an iterable for every two neighbouring elements.

Parameters
iterable (*Iterable | numpy.ndarray*) – An iterable variable

Returns
iterator that pairs every two neighbouring elements in the `iterable`

Return type
zip

Examples:

```
>>> from src.utils import paired_next
>>> lst = [1, 3, 5, 7, 9]
>>> list(paired_next(lst))
[(1, 3), (3, 5), (5, 7), (7, 9)]
```

src.utils.grouped

src.utils.**grouped**(*iterable*, *n*)

Group every two elements of an iterable variable.

Parameters

- **iterable** – An iterable variable.
- **iterable** – typing.Iterable
- **n** (*int*) – Number of elements to be grouped.

Returns
An iterator that groups every `n` elements of the `iterable`.

Return type
zip

Examples:

```
>>> from src.utils import grouped
>>> rng = range(0, 20, 2)
>>> list(grouped(rng, 2))
[(0, 2), (4, 6), (8, 10), (12, 14), (16, 18)]
```

src.utils.add_sql_query_elr_condition

src.utils.**add_sql_query_elr_condition**(*sql_query*, *elr=None*)

Add a condition about specific ELR(s) to a given SQL query statement.

Parameters

- **sql_query** (*str*) – SQL query statement.
- **elr** (*tuple | list | str | None*) – ELR; defaults to None.

Returns

Updated SQL query statement with specific ELR(s).

Return type

str

Examples:

```
>>> from src.utils import add_sql_query_elr_condition
>>> query = 'SELECT * FROM a_table'
>>> query
'SELECT * FROM a_table'
>>> add_sql_query_elr_condition(query)
'SELECT * FROM a_table'
>>> add_sql_query_elr_condition(query, elr='ECM8')
'SELECT * FROM a_table WHERE "ELR"='ECM8''
>>> add_sql_query_elr_condition(query, elr=['ECM7', 'ECM8'])
'SELECT * FROM a_table WHERE "ELR" IN ('ECM7', 'ECM8')'
```

src.utils.numba_np_shift

src.utils.**numba_np_shift**(*array*, *step*, *fill_value=nan*)

Shift an array by desired number of rows with Numba.

Parameters

- **array** (*numpy.ndarray*) – An array of numbers.
- **step** (*int*) – Number of rows to shift.
- **fill_value** (*float | int*) – Values to fill missing rows due to the shift; defaults to NaN.

Returns

Shifted array.

Return type

numpy.ndarray

Examples:

```
>>> from src.utils import numba_np_shift
>>> import numpy
>>> arr = numpy.array([[10, 13, 17],
...                    [20, 23, 27],
```

(continues on next page)

(continued from previous page)

```
...                    [15, 18, 22],
...                    [30, 33, 37],
...                    [45, 48, 52]], dtype='float32')
>>> numba_np_shift(arr, step=-1)
array([[20., 23., 27.],
       [15., 18., 22.],
       [30., 33., 37.],
       [45., 48., 52.],
       [nan, nan, nan]], dtype=float32)
>>> numba_np_shift(arr, step=1, fill_value=0)
array([[ 0.,  0.,  0.],
       [10., 13., 17.],
       [20., 23., 27.],
       [15., 18., 22.],
       [30., 33., 37.]], dtype=float32)
```

src.utils.find_valid_names

src.utils.**find_valid_names**(*name*, *valid_names*)

Find valid names.

Parameters

- **name** (*str | list | None*) – One or a sequence of str values.
- **valid_names** (*list*) – A list of valid str values.

Returns

Valid value(s).

Return type

list

Examples:

```
>>> from src.utils import find_valid_names
>>> valid_names = [
...     'LeftTopOfRail', 'LeftRunningEdge', 'RightTopOfRail', 'RightRunningEdge',
 ↪'Centre']
>>> find_valid_names('top left', valid_names)
['LeftTopOfRail']
>>> find_valid_names('right edge', valid_names)
['RightRunningEdge']
```

src.utils.fix_folium_float_image

src.utils.**fix_folium_float_image**(*path_to_m*)

Fix incorrect display of floating images (when utilising folium<=0.13.0).

Parameters

path_to_m (*str*) – Path to an HTML file saved by folium.

2.1.2 Utilities for geometric data

parse_projcs(path_to_projcs[, parser, as_dict])	Parse a PROJCS (Projected Coordinate Systems) data file.
flatten_geometry(geom[, as_array])	Flatten a series of geometry object.
calculate_slope(xy_arr[, method, na_val])	Calculate the slope for a given 2D array.
point_projected_to_line(point, line[, drop])	Find the projected point from a known point to a line.
make_a_polyline(points_sequence[, ...])	Sort a sequence of geographic coordinates to form a real path.
offset_ls(ls1, ls2[, cap_style, as_array])	Offset a linestring of longer length (`ls2`) by a shorter one (`ls1`).
geom_distance(geom_obj)	Get a key (function type) to search for relative distance.
extrapolate_line_point(polyline, dist[, ...])	Extrapolate the coordinates of a point that lies on extension of a given line at a given distance.

src.utils.parse_projcs

src.utils.**parse_projcs**(*path_to_projcs*, *parser='osr'*, *as_dict=False*)

Parse a PROJCS (Projected Coordinate Systems) data file.

This function reads and parses a .prj data file, which contains information about projected coordinate systems.

Parameters

- **path_to_projcs** (*str*) – The file path to a .prj data file.
- **parser** (*str*) – The name of the package used to read the .prj data file. Valid options are `{'osr', 'pycrs'}`; defaults to `'osr'`.
- **as_dict** (*bool*) – Whether to return the data as a dictionary; defaults to True.

Returns

The parsed data from the .prj file, either as a string or a dictionary.

Return type

str | dict

Raises

- **ValueError** – If the provided parser is not one of the valid options.
- **FileNotFoundError** – If the .prj file cannot be found at the given path.
- **Exception** – For any other errors during file parsing.

See also:

get_dgn_shp_prj() for examples of this function in use.

src.utils.flatten_geometry

src.utils.**flatten_geometry**(*geom*, *as_array=True*)

Flatten a series of geometry object.

Parameters

- **geom** (*pandas.Series*) – A series of geometry object.
- **as_array** (*bool*) – Whether to return the flattened geometry as an array; defaults to True.

Returns

An array of point coordinates.

Return type

list | numpy.ndarray

Examples:

```
>>> from src.utils import flatten_geometry, TrackFixityDB
>>> from src.preprocessor import PCD
>>> from pyhelpers.settings import np_preferences
>>> db_instance = TrackFixityDB()
>>> pcd = PCD(db_instance=db_instance)
>>> tbl_name = pcd.dgn_shp_table_name_('Polyline')
>>> query = f'SELECT * FROM "PCD"."{tbl_name}" WHERE "Year"=2020 AND "Month"=4 LIMIT␣
↪5'
>>> example_dat = pcd.db_instance.read_sql_query(query)
>>> example_dat['geometry']
0    LINESTRING Z (340183.475 674108.682 33.237, 34...
1    LINESTRING Z (340184.412 674109.031 33.235, 34...
2    LINESTRING Z (340185.349 674109.38 33.232, 340...
3    LINESTRING Z (340233.146 674127.169 33.117, 34...
4    LINESTRING Z (340234.083 674127.518 33.114, 34...
Name: geometry, dtype: object
>>> np_preferences()
>>> flatten_geometry(geom=example_dat.geometry)
array([[340183.4750, 674108.6820, 33.2370],
       [340183.4750, 674108.6820, 33.2550],
       [340184.4120, 674109.0310, 33.2350],
       [340184.4120, 674109.0310, 33.2530],
       [340185.3490, 674109.3800, 33.2320],
       [340185.3490, 674109.3800, 33.2500],
```

(continues on next page)

(continued from previous page)

```
       [340233.1460, 674127.1690, 33.1170],
       [340233.1460, 674127.1690, 33.1350],
       [340234.0830, 674127.5180, 33.1140],
       [340234.0830, 674127.5180, 33.1320]])
```

src.utils.calculate_slope

src.utils.**calculate_slope**(*xy_arr*, *method=None*, *na_val=None*, ***kwargs*)

Calculate the slope for a given 2D array.

Parameters

- **xy_arr** (*numpy.ndarray*) – A 2D array.
- **method** (*str | None*) – Method used to calculate the slope, and options include {'numpy', 'scipy'}; defaults to None.
- **na_val** (*numpy.nan | int | float | None*) – Replace NA/NaN (which is the result) with a specific value; defaults to None.

Returns

Slope of xy_arr.

Return type

float

Examples:

```
>>> from src.utils import calculate_slope
>>> import numpy
>>> arr1 = numpy.array([[399299.5160, 655099.1290], [399299.9990, 655098.2540]])
>>> slope1 = calculate_slope(arr1)
>>> slope1
-1.8116169544740974
>>> slope1_1 = calculate_slope(arr1, method='numpy')
>>> slope1_1
-1.8115942032905608
>>> slope1_2 = calculate_slope(arr1, method='scipy')
>>> slope1_2
-1.8115942028706058
>>> arr2 = numpy.array([[399299.9540, 655091.4080, 43.2010],
...                     [399299.4700, 655092.2840, 43.2060],
...                     [399298.9850, 655093.1580, 43.2110],
...                     [399298.5010, 655094.0330, 43.2170],
...                     [399298.0160, 655094.9070, 43.2230],
...                     [399297.5330, 655095.7830, 43.2290],
...                     [399297.0490, 655096.6580, 43.2330],
...                     [399296.5650, 655097.5330, 43.2390],
...                     [399296.0800, 655098.4080, 43.2430],
...                     [399295.5960, 655099.2830, 43.2480]])
>>> slope2 = calculate_slope(arr2)
>>> slope2
-1.807090900484403
>>> slope2_1 = calculate_slope(arr2, method='numpy')
```

(continues on next page)

(continued from previous page)

```
>>> slope2_1
-1.8070743791308161
>>> slope2_2 = calculate_slope(arr2, method='scipy')
>>> slope2_2
-1.8070743792147357
```

src.utils.point_projected_to_line

src.utils.**point_projected_to_line**(*point*, *line*, *drop=None*)

Find the projected point from a known point to a line.

Parameters

- **point** (*shapely.geometry.Point*) – Geometry object of a point.
- **line** (*shapely.geometry.LineString*) – Ggeometry object of a line.
- **drop** (*str | None*) – Which dimension to drop, and options include {'x', 'y', 'z'}; defaults to None.

Returns

The original point (with all or partial dimensions, given drop) and the projected one.

Return type

tuple

Examples:

```
>>> from src.utils import point_projected_to_line
>>> from shapely.geometry import Point, LineString
>>> pt = Point([399297, 655095, 43])
>>> ls = LineString([[399299, 655091, 42], [399295, 655099, 42]])
>>> _, pt_ = point_projected_to_line(pt, ls)
>>> pt_.wkt
'POINT Z (399297 655095 42)'
```

src.utils.make_a_polyline

src.utils.**make_a_polyline**(*points_sequence*, *start_point=None*, *reverse=False*, *as_geom=True*)

Sort a sequence of geographic coordinates to form a real path.

Parameters

- **points_sequence** (*shapely.geometry.LineString | list | numpy.ndarray*) – A sequence of geographic coordinates.
- **start_point** (*list | tuple | numpy.ndarray | shapely.geometry.Point | None*) – Start point of a path; defaults to None.

- **reverse** (*bool*) – Whether to reverse the resultant path; defaults to `False`.
- **as_geom** (*bool*) – Whether to return the sorted path as a line geometry object; defaults to `True`.

Returns
(Sorted) coordinates of the input `polyline`.

Return type
list | shapely.geometry.LineString

Examples:

```
>>> from src.utils import make_a_polyline
>>> polyline_coords = [(360100, 677100), (360000, 677100), (360200, 677100)]
>>> # polyline = polyline_coords.copy()
>>> # start_point = (360000, 677100)
>>> rslt = make_a_polyline(polyline_coords, start_point=(360000, 677100))
>>> rslt.wkt
'LINESTRING (360000 677100, 360100 677100, 360200 677100)'
```

src.utils.offset_ls

`src.utils.`**offset_ls**(*ls1*, *ls2*, *cap_style=1*, *as_array=False*)

Offset a linestring of longer length (`ls2`) by a shorter one (`ls1`).

Parameters

- **ls1** (*shapely.geometry.LineString | numpy.ndarray*) – A line.
- **ls2** (*shapely.geometry.LineString | numpy.ndarray*) – A line (shorter than `ls1`).
- **cap_style** (*int*) – `cap_style` enumerated by the object shapely.geometry.CAP_STYLE; defaults to `1`.
- **as_array** (*bool*) – Whether to return the result as numpy.ndarray type; defaults to `False`.

Returns
`ls1` and shortened `ls2`.

Return type
tuple

Examples:

```
>>> from src.utils import offset_ls
>>> from pyhelpers.settings import np_preferences
>>> import numpy
>>> np_preferences()
>>> x1 = numpy.arange(start=0, stop=10, step=0.1)
>>> y1 = x1 + 1
>>> x2 = numpy.arange(start=1, stop=20, step=0.5)
```

(continues on next page)

(continued from previous page)

```
>>> y2 = 2 * x2 + 2
>>> l1_arr, l2_arr = map(numpy.column_stack, ((x1, y1), (x2, y2)))
>>> l1_1, l2_1 = offset_ls(ls1=l1_arr, ls2=l2_arr, as_array=True)
>>> l2_1
array([[1.0000, 4.0000],
       [1.5000, 5.0000],
       [2.0000, 6.0000],
       [2.5000, 7.0000],
       [3.0000, 8.0000],
       [3.5000, 9.0000],
       [4.0000, 10.0000],
       [4.5000, 11.0000],
       [5.0000, 12.0000],
       [5.5000, 13.0000],
       [5.6976, 13.3953]])
>>> l1_2, l2_2 = offset_ls(ls1=l1_arr, ls2=l2_arr, cap_style=2, as_array=True)
>>> l2_2
array([[1.0000, 4.0000],
       [1.5000, 5.0000],
       [2.0000, 6.0000],
       [2.5000, 7.0000],
       [3.0000, 8.0000],
       [3.5000, 9.0000],
       [4.0000, 10.0000],
       [4.5000, 11.0000],
       [5.0000, 12.0000],
       [5.5000, 13.0000],
       [6.0000, 14.0000],
       [6.2667, 14.5333]])
```

Illustration:

```
import matplotlib.pyplot as plt
from pyhelpers.settings import mpl_preferences
mpl_preferences(backend='TkAgg')
fig = plt.figure()
ax = fig.add_subplot()
ax.plot(x1, y1, label='ls1')
ax.plot(x2, y2, label='ls2')
# cap_style=1
ax.plot(l2_1[:, 0], l2_1[:, 1], label='ls2 (offset; cap_style=1)', linewidth=8)
# cap_style=2
ax.plot(l2_2[:, 0], l2_2[:, 1], label='ls2 (offset; cap_style=2)', linewidth=4)
ax.legend(loc='best')
fig.tight_layout()
# fig.savefig("docs\source\_images\utils_offset_ls_demo.svg")
# fig.savefig("docs\source\_images\utils_offset_ls_demo.pdf")
```

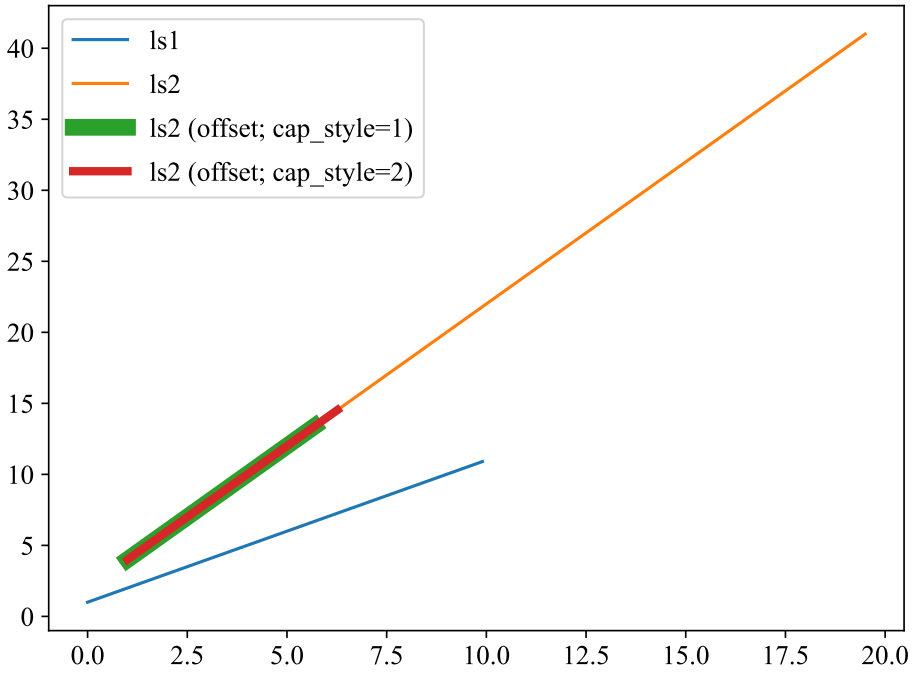


Figure 3: An example of a line ('ls2') offset by another ('ls1').

src.utils.geom_distance

src.utils.**geom_distance**(*geom_obj*)

Get a key (function type) to search for relative distance.

Parameters

geom_obj – A geometry object.

Type

shapely.geometry.base.BaseGeometry

Returns

The 'key' to search for relative distance.

Return type

functools.partial

See also:

Examples of the method *TrackMovement.calculate_unit_subsection_displacement()*.

src.utils.extrapolate_line_point

src.utils.**extrapolate_line_point**(*polyline*, *dist*, *deg=1*, *reverse=False*, *as_geom=True*)

Extrapolate the coordinates of a point that lies on extension of a given line at a given distance.

Parameters

- **polyline** (*shapely.geometry.LineString*) – A polynomial line
- **dist** (*float | int*) – distance (in metre) beyond an end of the given line
- **deg** (*int*) – degree of the polynomial line; defaults to `1`
- **reverse** (*bool*) – indicate the direction of th extrapolation; defaults to `False`
- **as_geom** (*bool*) – Whether to return as a shapely geometry object; defaults to `True`

Returns

A point that extends a given line at a given distance

Return type

tuple | shapely.geometry.Point

Examples:

```
>>> from src.utils import extrapolate_line_point
>>> from pyhelpers.settings import np_preferences
>>> from shapely.geometry import LineString
>>> import numpy
>>> np_preferences()
>>> ls_arr = numpy.array([[326301.3395, 673958.8021, 0.0000],
...                       [326302.2164, 673959.0613, 0.0000],
...                       [326309.2315, 673961.1349, 0.0000]])
>>> ls = LineString(ls_arr)
>>> pt = extrapolate_line_point(ls, dist=1, deg=1, reverse=True)
>>> pt.wkt
'POINT (326300.3805232464 673958.5186338174)'
```

Illustration:

```
import matplotlib.pyplot as plt
from pyhelpers.settings import mpl_preferences
mpl_preferences(backend='TkAgg')
fig = plt.figure()
ax = fig.add_subplot()
ax.ticklabel_format(useOffset=False)
ax.plot(ls_arr[:, 0], ls_arr[:, 1], label='A line')
ax.scatter(pt.x, pt.y, label='The point being extrapolated')
ax.legend()
fig.tight_layout()
# fig.savefig("docs\source\_images\utils_extrapolate_line_point_demo.svg")
# fig.savefig("docs\source\_images\utils_extrapolate_line_point_demo.pdf")
```

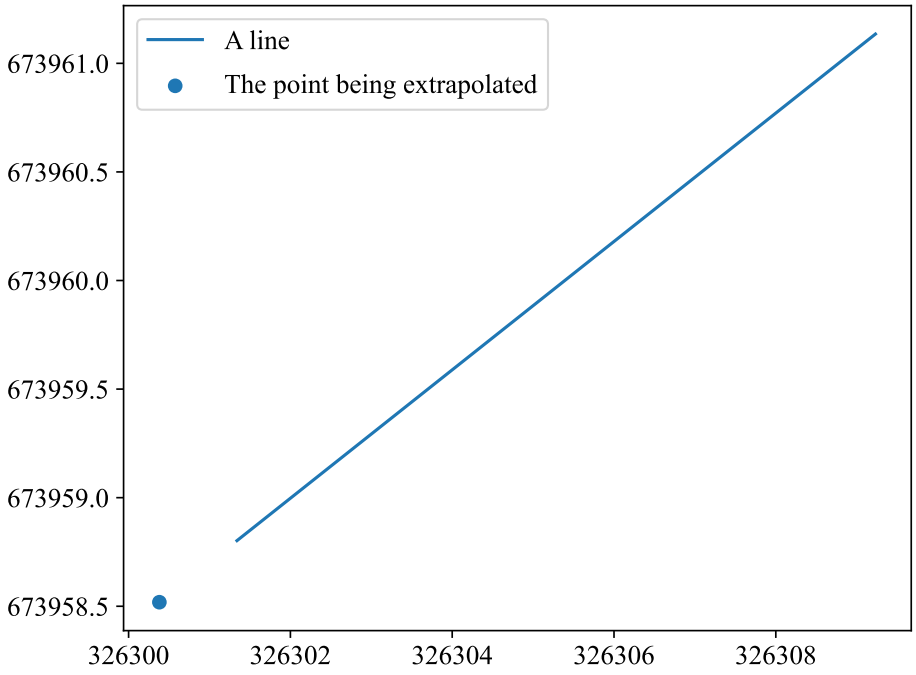


Figure 4: An example of a point being extrapolated from a given line.

2.1.3 Utilities for DGN data and shapefiles

get_dgn_shp_prj(path_to_file[, ...])	Read metadata of projection for DGN files or shapefiles.
dgn_shapefiles()	Get all types of shapefiles converted from a DGN file.
dgn_shapefile_ext()	Get all file extensions of shapefiles converted from a DGN file.
remove_dgn_shapefiles(path_to_dgn)	Remove all shapefiles converted from a .dgn file.
dgn2shp(path_to_dgn[, output_dir, ...])	Convert a DGN file into shapefiles.
dgn2shp_batch(dat_name, file_paths[, ...])	Convert a list of DGN files to shapefiles.
get_field_names(path_to_file)	Get field names for a specific item.
read_dgn_shapefile(path_to_dir, dgn_filename)	Read the shapefiles converted from a DGN data file.
dgn_shp_map_view(cls_instance, item_name, ...)	Get a map view of a specific item.

src.utils.get_dgn_shp_prj

src.utils.**get_dgn_shp_prj**(*path_to_file*, *projcs_parser='osr'*, *as_dict=True*, *update=False*, *verbose=False*)

Read metadata of projection for DGN files or shapefiles.

Parameters

- **path_to_file** (*str*) – Path to a PROJCS file (without the .prj file extension).
- **projcs_parser** (*str*) – Name of the package used to read the .prj data file. Valid options are {'osr', 'pycrs'}; defaults to 'osr'.
- **as_dict** (*bool*) – Whether to return the data as a dictionary; defaults to True.
- **update** (*bool*) – Whether to read the original data instead of loading from a pickle file if available; defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console as the function runs; defaults to False.

Returns

Parsed PROJCS data and shape boundary. Returns None if parsing fails.

Return type

dict | None

Raises

- **ValueError** – If the provided parser is not one of the valid options.
- **FileNotFoundError** – If the .prj file cannot be found at the given path.
- **Exception** – For any other errors during file parsing.

Examples:

```
>>> from src.utils import get_dgn_shp_prj
>>> from pyhelpers.dirs import cd
>>> from src.preprocessor import OPAS
>>> opas = OPAS()
>>> file_path = cd(opas.DATA_DIR, opas.PROJ_DIRNAME, opas.PROJ_FILENAME)
>>> prj_dat = get_dgn_shp_prj(path_to_file=file_path)
>>> type(prj_dat)
dict
>>> list(prj_dat.keys())
['PROJCS', 'Shapefile']
>>> list(prj_dat['PROJCS'].keys())
['proj', 'lat_0', 'lon_0', 'k', 'x_0', 'y_0', 'ellps', 'units']
>>> prj_dat['Shapefile'].shape
(1, 4)
>>> prj_dat = get_dgn_shp_prj(path_to_file=file_path, as_dict=False)
>>> prj_dat['PROJCS'][:11]
'+proj=tmerc'
```

src.utils.dgn_shapefiles

src.utils.**dgn_shapefiles**()

Get all types of shapefiles converted from a DGN file.

Returns
A list of shapefile types that can be converted from a DGN (.dgn) file.

Return type
list[str]

Examples:

```
>>> from src.utils import dgn_shapefiles
>>> dgn_shapefiles()
['Annotation', 'MultiPatch', 'Point', 'Polygon', 'Polyline']
```

src.utils.dgn_shapefile_ext

src.utils.**dgn_shapefile_ext**()

Get all file extensions of shapefiles converted from a DGN file.

Returns
A list of file extensions for shapefiles converted from a DGN (.dgn) file.

Return type
list[str]

Examples:

```
>>> from src.utils import dgn_shapefile_ext
>>> dgn_shapefile_ext()
['.cpg', '.dbf', '.sbn', '.sbx', '.shp', '.shp.xml', '.shx']
```

src.utils.remove_dgn_shapefiles

src.utils.**remove_dgn_shapefiles**(*path_to_dgn*)

Remove all shapefiles converted from a .dgn file.

Parameters
path_to_dgn (*str*) – The absolute path to the original .dgn data file.

See also:

dgn2shp_batch() for examples of this function in use.

src.utils.dgn2shp

src.utils.**dgn2shp**(*path_to_dgn*, *output_dir=None*, *export_to='shapefile'*, *verbose=False*, *python2=None*)

Convert a DGN file into shapefiles.

The output includes a set of files with extensions such as .cpg, .dbf, .sbn, .sbx, .shp, .shp.xml, and .shx. The data is categorised into annotation, multipatch, point, polygon and polyline.

This function uses the arcpy package available in ArcGIS Desktop.

See also [UD-1] and [UD-2].

Parameters

- **path_to_dgn** (*str*) – The path to the .dgn file.
- **output_dir** (*str | None*) – The absolute path to the folder where output files will be saved. When `output_dir=None` (default), the files are saved in the same directory as the .dgn file.
- **export_to** (*str*) – The format to export the data to; defaults to `'shapefile'`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console as the function runs; defaults to `False`.
- **python2** (*str*) – The path to the Python 2.7 executable. When `python2=None` (default), it defaults to `"C:\Python27\ArcGIS10.7\python"`.

Examples:

```
>>> from src.utils import dgn2shp
>>> import glob
>>> from pyhelpers.dirs import cd, cdd
>>> temp_dat_dir = cdd("temp", "CARRS", "Tunnels")
>>> opas_stn_dgn = cd(temp_dat_dir, "ENG_ADMIN.RINM_CARRS_TunnelPortal.dgn")
>>> dgn2shp(opas_stn_dgn, verbose=True)
Converting "ENG_ADMIN.RINM_CARRS_TunnelPortal.dgn" ... ... Done.
>>> glob.glob1(temp_dat_dir, "*.shp")
['ENG_ADMIN_RINM_CARRS_TunnelPortal_dgn_Annotation.shp',
 'ENG_ADMIN_RINM_CARRS_TunnelPortal_dgn_MultiPatch.shp',
 'ENG_ADMIN_RINM_CARRS_TunnelPortal_dgn_Point.shp',
 'ENG_ADMIN_RINM_CARRS_TunnelPortal_dgn_Polygon.shp',
 'ENG_ADMIN_RINM_CARRS_TunnelPortal_dgn_Polyline.shp']
```

src.utils.dgn2shp_batch

src.utils.**dgn2shp_batch**(*dat_name*, *file_paths*, *confirmation_required=True*, *verbose=False*, ***kwargs*)

Convert a list of DGN files to shapefiles.

Parameters

- **dat_name** (*str*) – Name of the data category.
- **file_paths** (*list [str]*) – A list of paths to DGN files.
- **confirmation_required** (*bool*) – Whether to prompt for confirmation before proceeding; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console as the function runs; defaults to `False`.
- **kwargs** – [Optional] parameters of the *dgn2shp()* function.

See also:

`dgn2shp()` for examples of the method.

src.utils.get_field_names

src.utils.**get_field_names**(*path_to_file*)

Get field names for a specific item.

Parameters
path_to_file (*str*) – Path to the file that contains field names.

Returns
A list of field names.

Return type
list

Examples:

```
>>> from src.utils import get_field_names
>>> from pyhelpers.dirs import cdd
>>> file_path = cdd("CARRS", "Overline bridges", "FID_Point.txt")
>>> fn = get_field_names(file_path)
>>> type(fn)
list
>>> fn[0:10]
['Entity',
 'Handle',
 'Level',
 'Layer',
 'LvlDesc',
 'LyrFrzn',
 'LyrLock',
 'LyrOn',
```

(continues on next page)

(continued from previous page)

```
'LvlPlot',
'Color']
```

src.utils.read_dgn_shapefile

src.utils.**read_dgn_shapefile**(*path_to_dir*, *dgn_filename*)

Read the shapefiles converted from a DGN data file.

Parameters

- **path_to_dir** (*str*) – Path to the data directory.
- **dgn_filename** (*str*) – Filename of the original DGN data.

Returns

Data of the converted shapefiles.

Return type

dict

See also:

Examples for the following methods of the class *CARRS*:

- *read_overline_bridges_shp()*
- *read_underline_bridges_shp()*
- *read_retaining_walls_shp()*
- *read_tunnels_shp()*

src.utils.dgn_shp_map_view

src.utils.**dgn_shp_map_view**(*cls_instance*, *item_name*, *layer_name*, *desc_col_name*, *sample=True*, *marker_colour='blue'*, *update=False*, *verbose=True*)

Get a map view of a specific item.

Parameters

- **cls_instance** (src.preprocessor.CARRS | src.preprocessor.CNM | src.preprocessor.OPAS) – Instance of a data class.
- **item_name** (*str*) – Name of an item.
- **layer_name** (*str*) – Name of a layer.
- **desc_col_name** (*str*) – Name of a column that describes markers.
- **sample** (*bool* | *int*) – Whether to draw a sample, or a specific sample size.

- **marker_colour** (*str*) – Colour of markers; defaults to `'blue'`.
- **update** (*bool*) – Whether to read the original data (instead of loading its pickle file if available); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in console as the function runs; defaults to `True`.

See also:

Examples for the following methods:

- *CARRS.map_view()*
- *CNM.map_view()*
- *OPAS.map_view()*

2.1.4 Utilities for LAZ/LAS data

download_las_tools([las_tool, download_dir, ...])	Download a LAS tool from https://lastools.github.io/.
laz2las(laz_files[, output_las, ...])	Decompress a LAZ file to a LAS file.
merge_las_files(las_files[, output_las, ...])	Merge multiple LAS files into a single LAS file.

src.utils.download_las_tools

src.utils.**download_las_tools**(*las_tool=None*, *download_dir=None*, *sub_dir=''*, *update=False*, *verbose=True*, *ret_tool_path=False*, ***kwargs*)

Download a LAS tool from https://lastools.github.io/.

Parameters

- **las_tool** (*str | None*) – Name of the LAS tool to be downloaded; options include `{'LAStools.zip', 'laszip.exe', 'laszip64.exe', 'laszip-cli.exe'}`; when `las_tool=None` (default), it is to use/download "laszip64.exe".
- **download_dir** (*str | None*) – Name of a directory for the downloaded LAS tool; defaults to `None`.
- **sub_dir** (*str*) – Name of a subdirectory; defaults to `""`.
- **update** (*bool*) – Whether to update the existing data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in console as the function runs; defaults to `True`.
- **ret_tool_path** (*bool*) – Whether to return the absolute path to the downloaded file; defaults to `False`.

- **kwargs** – [Optional] parameters of pyhelpers.ops.download_file_from_url.

Returns

An absolute path to the downloaded file (only when `ret_tool_path=True`).

Return type

str

Examples:

```
>>> from src.utils import download_las_tools
>>> import os
>>> from pyhelpers.dirs import cdd
>>> dwnld_dir = cdd("PCD")
>>> # # download_las_tools('laszip', dwnld_dir, update=True, verbose=True)
>>> download_las_tools('laszip', dwnld_dir)
Downloading "laszip.exe" ... Done.
>>> # "laszip.exe" already exists at "data\PCD"
>>> path_to_laszip64 = download_las_tools('laszip64', dwnld_dir, ret_tool_path=True)
"laszip64.exe" already exists at "data\PCD"
>>> os.path.relpath(path_to_laszip64)
'data\PCD\laszip64.exe'
```

src.utils.laz2las

src.utils.**laz2las**(*laz_files*, *output_las=None*, *temp_file_list=False*, *laszip_exe='laszip64.exe'*)

Decompress a LAZ file to a LAS file.

Parameters

- **laz_files** (*str*) – Path(s) to source LAZ file(s).
- **output_las** (*str*) – Path to a LAS file as the output; defaults to None.
- **temp_file_list** (*bool*) – Whether to use a temporary .txt file to list all LAZ files; defaults to False.
- **laszip_exe** (*str*) – Path to the executable file laszip64; defaults to "laszip64.exe".

See also [L2L-1] and [L2L-2].

Examples:

```
>>> from src.utils import laz2las
>>> import os
>>> from pyhelpers.dirs import cd, cdd
>>> pcd_dir = cdd("PCD")
>>> laz_dir = cd(pcd_dir, "ECM8\LAZ_OSGB_100x100\201910")
>>> filename = "Tile_X+0000340100_Y+0000674000.laz"
>>> laszip_exe_ = cd(pcd_dir, "laszip.exe")
```

(continues on next page)

(continued from previous page)

```
>>> # -- Example 1 ------------------------------------------------------------
 ↪-----
>>> laz_file = cd(laz_dir, filename)  # path to the source LAZ file
>>> las_file = laz_file.replace(".laz", ".las")  # path to the output LAS file
>>> # Decompress the (source) LAZ file to the (output) LAS file
>>> laz2las(laz_file, las_file, laszip_exe=laszip_exe_)

>>> # -- Example 2 ------------------------------------------------------------
 ↪-----
>>> laz_wild_card = os.path.relpath(cd(laz_dir, "*.laz"))  # paths to the source LAZ␣
 ↪files
>>> laz2las(laz_wild_card, laszip_exe=laszip_exe_)
```

src.utils.merge_las_files

src.utils.**merge_las_files**(*las_files*, *output_las=None*, *laszip_exe='laszip64.exe'*, *ret_output_path=False*)

Merge multiple LAS files into a single LAS file.

Parameters

- **las_files** (*list | str*) – Paths to LAS files or a path to a .txt file that contains a list of LAS files.
- **output_las** (*str | None*) – Path to a LAS file; defaults to None.
- **laszip_exe** (*str*) – Path to the laszip executable; defaults to "laszip64.exe".
- **ret_output_path** (*bool*) – Whether to return the path to the output merged LAS data file

Returns

(Only if ret_output_path=True) an absolute path to the output merged LAS data file.

Return type

str

See also [L2L-1] and [L2L-2].

Examples:

```
>>> from src.utils import laz2las, merge_las_files
>>> import os
>>> from pyhelpers.dirs import cd, cdd
>>> laszip_exe_ = cdd("PCD\laszip.exe")
>>> pcd_dir = cdd("PCD")
>>> laz_dir = cd(pcd_dir, "ECM8\LAZ_OSGB_100x100")
>>> # Decompress LAZ files to LAS files
>>> laz_wild_card = os.path.relpath(cd(laz_dir, "*.laz"))
>>> laz2las(laz_wild_card, laszip_exe=laszip_exe_)
>>> # Merge the LAS files into a LAS file
>>> las_wild_card = os.path.relpath(cd(laz_dir, "*.las"))
```

(continues on next page)

(continued from previous page)

```
>>> merged_file = merge_las_files(
...     las_wild_card, laszip_exe=laszip_exe_, ret_output_path=True)
>>> os.path.relpath(merged_file, pcd_dir)
'ECM8\LAZ_OSGB_100x100\LAZ_OSGB_100x100.las'
```

2.1.5 Error classes

InvalidSubsectionLength	Indicate invalid subsection length.

InvalidSubsectionLength

class src.utils.**InvalidSubsectionLength**

Indicate invalid subsection length.

This class is utilised in the method *TrackMovement.split_section()*. The error would be raised if the `unit_length` exceeds the length of `section`.

Note: No directly defined attributes. See inherited class attributes.

Note: No directly defined methods. See inherited class methods.

2.2 preprocessor

Preprocessing and I/O management of all the data resources that are made available for the project.

Classes

Ballast([db_instance])	*Ballast*.
CARRS([db_instance, elr])	*Civil Asset Register and Reporting System*.
CNM([db_instance])	*Corporate Network Model*.
Geology([db_instance])	*Geology*.
GPR([db_instance])	*Ground Penetrating Radar*.
INM([db_instance])	*Integrated Network Model*.
OPAS([db_instance])	*Operational Property Asset System*.
PCD([elr, db_instance])	*Point cloud data*.
Reports([db_instance])	Spreadsheet-based reports.
Track([db_instance])	*Track layout and quality*.

2.2.1 Ballast

class src.preprocessor.**Ballast**(*db_instance=None*)

Ballast.

This class focuses on preprocessing summary data specific to ballast. It provides methods to read, import and load ballast summary data from local directories or the project database, facilitating data cleaning, transformation and preparation for further analysis or modelling.

Note: This class currently handles with only a summary data about the ballasts.

Parameters

db_instance (TrackFixityDB | *None*) – Optional PostgreSQL database instance; defaults to None.

Variables

- **object_dtype** (*dict*) – Object field types in the ballast summary data.
- **float_dtype** (*dict*) – Float field types in the ballast summary data.
- **int_dtype** (*dict*) – Integer field types in the ballast summary data.
- **var_dtype** (*dict*) – Variable/mixed field types in the ballast summary data.
- **dtypes** (*dict*) – Field types of the ballast summary data.
- **db_instance** (TrackFixityDB) – PostgreSQL database instance used for data retrieval and storage.

Examples:

```
>>> from src.preprocessor import Ballast
>>> blst = Ballast()
>>> blst.NAME
'Ballast summary'
```

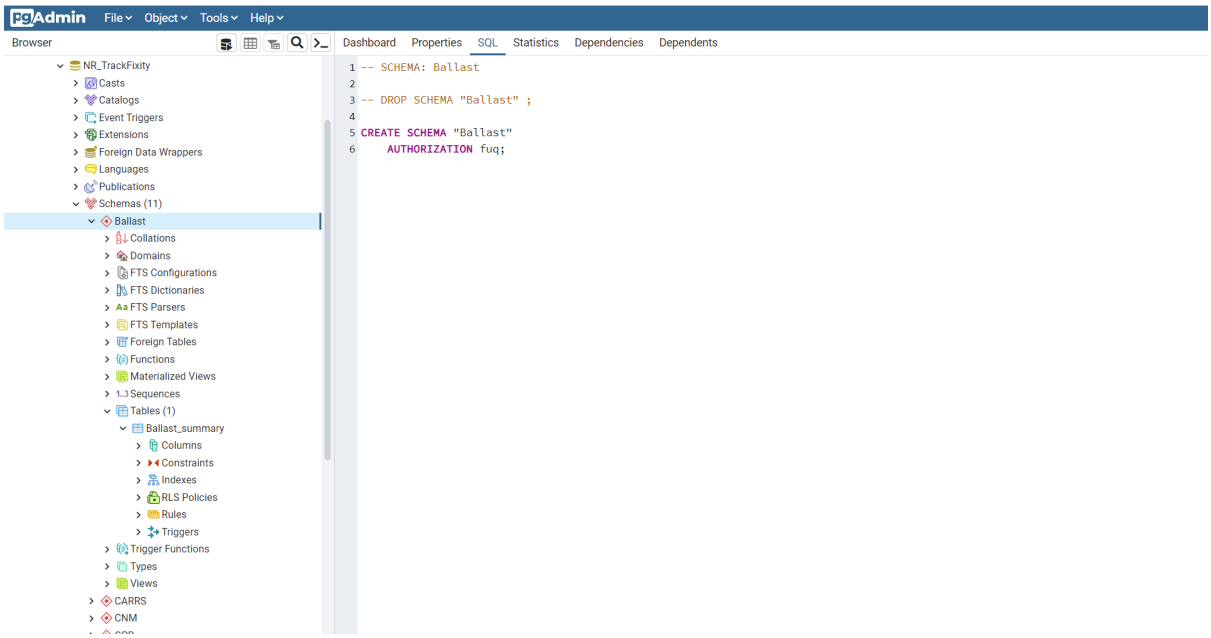


Figure 5: Snapshot of the *Ballast* schema.

Attributes:

DATA_DIR	Pathname of a local directory where the ballast data is stored.
NAME	Data name.
SCHEMA_NAME	Name of the schema for storing the ballast summary data.
SUM_FILENAME	Filename of the ballast summary data.
SUM_TABLE_NAME	Name of the table for storing the ballast summary data.

Ballast.DATA_DIR

Ballast.**DATA_DIR: str = 'data\\Ballast'**

Pathname of a local directory where the ballast data is stored.

Ballast.NAME

Ballast.**NAME: str = 'Ballast'**

Data name.

Ballast.SCHEMA_NAME

Ballast.**SCHEMA_NAME: str = 'Ballast'**

Name of the schema for storing the ballast summary data.

Ballast.SUM_FILENAME

Ballast.**SUM_FILENAME: str = 'Ballast summary'**

Filename of the ballast summary data.

Ballast.SUM_TABLE_NAME

Ballast.**SUM_TABLE_NAME: str = 'Ballast_summary'**

Name of the table for storing the ballast summary data.

Methods:

import_data([update, confirmation_required, ...])	Import ballast summary data into the project database.
load_data([elr, column_names, fmt_nr_mileage])	Load ballast summary data from the project database.
read_data([update, verbose])	Read ballast summary data from a local directory.

Ballast.import_data

Ballast.**import_data**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import ballast summary data into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **confirmation_required** (*bool*) – Whether a confirmation is required to proceed; defaults to True.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to True.
- **kwargs** – [Optional] additional parameters for the pyhelpers.dbms.PostgreSQL.import_data method.

Examples:

```
>>> from src.preprocessor import Ballast
>>> blst = Ballast()
>>> blst.import_data(if_exists='replace')
To import data of ballast summary into the table "Ballast"."Ballast_summary"?
 [No]|Yes: yes
Importing the data ... Done.
```

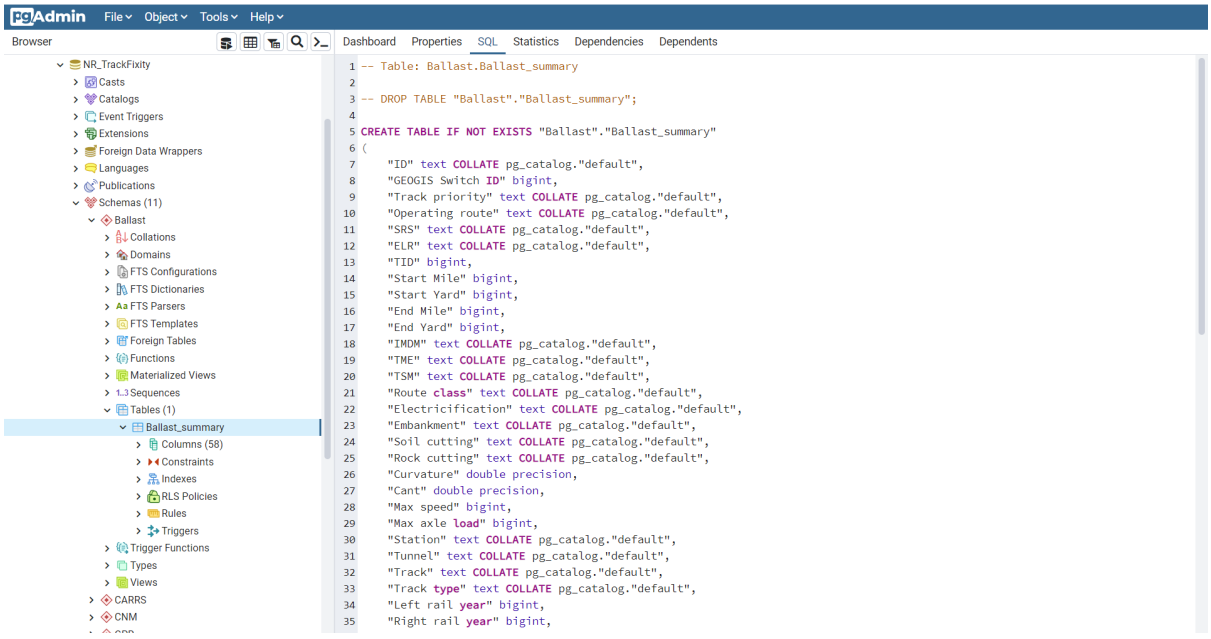


Figure 6: Snapshot of the "Ballast"."Ballast_summary" table.

Ballast.load_data

`Ballast.`**`load_data`**(*elr=None*, *column_names=None*, *fmt_nr_mileage=True*, ***kwargs*)

Load ballast summary data from the project database.

Parameters

- **elr** (*str | list | tuple | None*) – Engineer's Line Reference (ELR); defaults to `None`.
- **column_names** (*str | list | None*) – Names of columns (a subset) to be queried; defaults to `None`.
- **fmt_nr_mileage** (*bool*) – Whether to format Network Rail mileage data; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.read_sql_query.

Returns

Dictionary containing the loaded ballast summary data.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Ballast
>>> blst = Ballast()
>>> elr = ['ECM7', 'ECM8']
>>> data_ecm7_8 = blst.load_data(elr, fmt_nr_mileage=False)
>>> data_ecm7_8.shape
(8184, 58)
>>> data_ecm7_8_ = blst.load_data(elr, fmt_nr_mileage=True)
>>> data_ecm7_8_.shape
(8184, 60)
>>> data_ecm7_8_[['StartMileage', 'EndMileage']].head()
   StartMileage  EndMileage
0        0.0000      0.0054
1        0.0054      0.0060
2        0.0060      0.0066
3        0.0066      0.0084
4        0.0084      0.0092
```

Ballast.read_data

Ballast.**read_data**(*update=False*, *verbose=False*)

Read ballast summary data from a local directory.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns

DataFrame containing the ballast summary data, or None if data cannot be read.

Return type

pandas.DataFrame | None

Examples:

```
>>> from src.preprocessor import Ballast
>>> blst = Ballast()
>>> data = blst.read_data()
>>> data.shape
(650867, 58)
```

2.2.2 CARRS

class src.preprocessor.**CARRS**(*db_instance=None*, *elr=None*)

Civil Asset Register and Reporting System.

This class handles preprocessing of summary and DGN data for various types of structures, including overline *bridges*, *underline bridges*, *retaining walls* and *tunnels*.

Note: This class focuses specifically on handling summary and DGN data related to the aforementioned structure types.

Parameters

- **db_instance** (*TrackFixityDB | None*) – PostgreSQL database instance; defaults to None.
- **elr** (*str | list | tuple | None*) – Engineer's Line References; defaults to None.

Variables

- **elr** (*str*) – Engineer's Line References.
- **struct_dtypes** (*dict*) – Data types of the *structures* data.
- **ol_bdg_dgn_pathnames** (*list*) – Pathname(s) of DGN data of *overline bridges*.
- **ul_bdg_dgn_pathnames** (*list*) – Pathname(s) of DGN data of *underline bridges*.
- **rw_dgn_pathnames** (*list*) – Pathname(s) of DGN data of *retaining walls*.
- **tunl_dgn_pathnames** (*list*) – Pathname(s) of DGN data of *tunnels*.
- **db_instance** (*pyhelpers.dbms.PostgreSQL*) – PostgreSQL database instance used for data operations.

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> carrs.NAME
'Civil Asset Register and Reporting System'
```

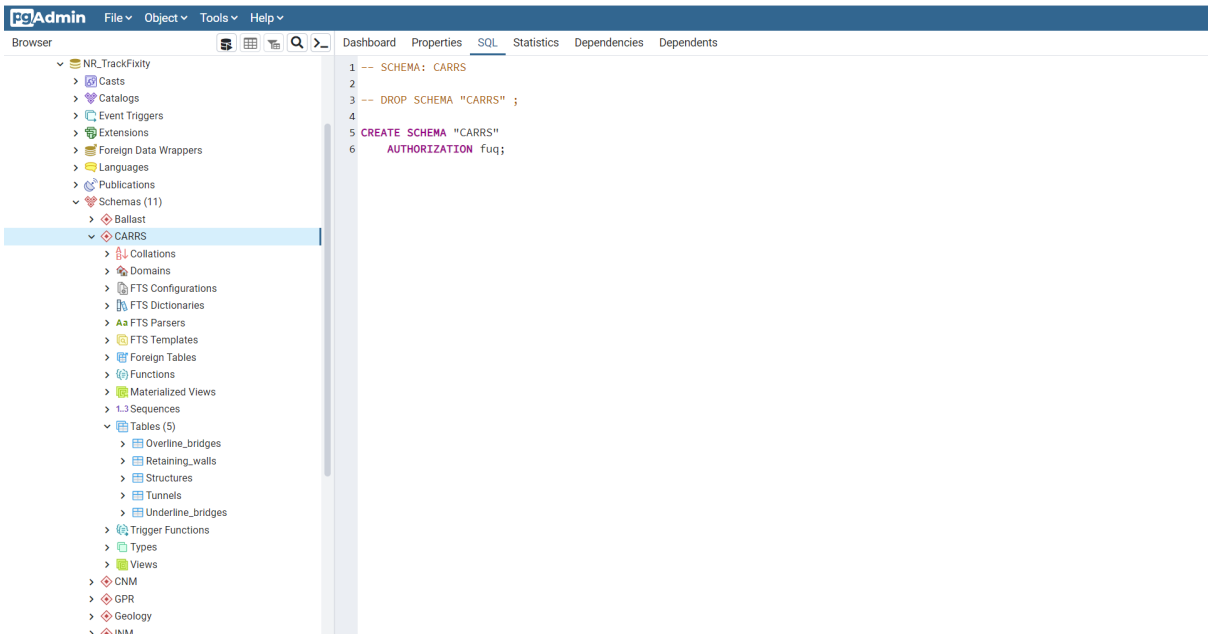


Figure 7: Snapshot of the *CARRS* schema.

Attributes:

ACRONYM	Acronym of the data name.
DATA_DIR	Pathname of a local directory where the CARRS data is stored.
NAME	Data name.
OL_BDG_DIRNAME	Directory name of *overline bridges* data.
OL_BDG_TABLE_NAME	Name of the table for storing the *overline bridges* data.
PROJ_DIRNAME	Directory name of projection/boundary shapefiles.
PROJ_FILENAME	Filename of the projection file.
RW_DIRNAME	Directory name of *retaining walls* data.
RW_TABLE_NAME	Name of the table for storing the *retaining walls* data.
SCHEMA_NAME	Name of the schema for storing the CARRS data.
STRUCT_DIRNAME	Directory name of *structures* data.
STRUCT_TABLE_NAME	Name of the table for storing the *structures* data.
TUNL_DIRNAME	Directory name of *tunnels* data.
TUNL_TABLE_NAME	Name of the table for storing the *tunnels* data.
UL_BDG_DIRNAME	Directory name of *underline bridges* data.
UL_BDG_TABLE_NAME	Name of the table for storing the *underline bridges* data.

CARRS.ACRONYM

CARRS.**ACRONYM: str = 'CARRS'**

Acronym of the data name.

CARRS.DATA_DIR

CARRS.**DATA_DIR: str = 'data\\CARRS'**

Pathname of a local directory where the CARRS data is stored.

CARRS.NAME

CARRS.**NAME: str = 'Civil Asset Register and Reporting System'**

Data name.

CARRS.OL_BDG_DIRNAME

CARRS.**OL_BDG_DIRNAME: str = 'Overline bridges'**

Directory name of *overline bridges* data.

CARRS.OL_BDG_TABLE_NAME

CARRS.**OL_BDG_TABLE_NAME: str = 'Overline_bridges'**

Name of the table for storing the *overline bridges* data.

CARRS.PROJ_DIRNAME

CARRS.**PROJ_DIRNAME: str = 'Projection'**

Directory name of projection/boundary shapefiles.

CARRS.PROJ_FILENAME

CARRS.**PROJ_FILENAME: str = 'Order_69433_Polygon'**

Filename of the projection file.

CARRS.RW_DIRNAME

CARRS.**RW_DIRNAME: str = 'Retaining walls'**

Directory name of *retaining walls* data.

CARRS.RW_TABLE_NAME

CARRS.**RW_TABLE_NAME: str = 'Retaining_walls'**

Name of the table for storing the *retaining walls* data.

CARRS.SCHEMA_NAME

CARRS.**SCHEMA_NAME: str = 'CARRS'**

Name of the schema for storing the CARRS data.

CARRS.STRUCT_DIRNAME

CARRS.**STRUCT_DIRNAME: str = 'Structures'**

Directory name of *structures* data.

CARRS.STRUCT_TABLE_NAME

CARRS.**STRUCT_TABLE_NAME: str = 'Structures'**

Name of the table for storing the *structures* data.

CARRS.TUNL_DIRNAME

CARRS.**TUNL_DIRNAME: str = 'Tunnels'**

Directory name of *tunnels* data.

CARRS.TUNL_TABLE_NAME

CARRS.**TUNL_TABLE_NAME: str = 'Tunnels'**

Name of the table for storing the *tunnels* data.

CARRS.UL_BDG_DIRNAME

CARRS.**UL_BDG_DIRNAME: str = 'Underline bridges'**

Directory name of *underline bridges* data.

CARRS.UL_BDG_TABLE_NAME

CARRS.**UL_BDG_TABLE_NAME: str = 'Underline_bridges'**

Name of the table for storing the *underline bridges* data.

Methods:

dgn2shp([dat_name, confirmation_required, ...])	Convert DGN files to shapefiles.
import_overline_bridges_shp(**kwargs)	Import shapefile data (converted from the DGN data) of *overline bridges* into the project database.
import_retaining_walls_shp(**kwargs)	Import shapefile data (converted from the DGN data) of *retaining walls* into the project database.
import_structures([update, ...])	Import data of structures into the project database.
import_tunnels_shp(**kwargs)	Import shapefile data (converted from the DGN data) of *tunnels* into the project database.
import_underline_bridges_shp(**kwargs)	Import shapefile data (converted from the DGN data) of *underline bridges* into the project database.
load_overline_bridges_shp(**kwargs)	Load the shapefile data (converted from the DGN data) of *overline bridges* from the project database.
load_retaining_walls_shp(**kwargs)	Load the shapefile data (converted from the DGN data) of *retaining walls* from the project database.
load_tunnels_shp(**kwargs)	Load the shapefile data (converted from the DGN data) of *tunnels* from the project database.
load_underline_bridges_shp(**kwargs)	Load the shapefile data (converted from the DGN data) of *underline bridges* from the project database.
map_view(structure_name[, desc_col_name, ...])	Make a map view of a CARRS item.
read_overline_bridges_shp(**kwargs)	Read the shapefile (converted from the DGN data) of *overline bridges* from a local directory.
read_prj_metadata([update, parser, as_dict, ...])	Read metadata of projection for the DGN files/shapefiles.
read_retaining_walls_shp(**kwargs)	Read the shapefile (converted from the DGN data) of *retaining walls* from a local directory.
read_structures([update, verbose])	Read data of structures from a local directory.
read_tunnels_shp(**kwargs)	Read the shapefile (converted from the DGN data) of *tunnels* from a local directory.
read_underline_bridges_shp(**kwargs)	Read the shapefile (converted from the DGN data) of *underline bridges* from a local directory.

CARRS.dgn2shp

CARRS.**dgn2shp**(*dat_name=None*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Convert DGN files to shapefiles.

Parameters

- **dat_name** (*str | None*) – Name of the data; defaults to None.
- **confirmation_required** (*bool*) – Whether a confirmation is required to proceed; defaults to True.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to True.
- **kwargs** – [Optional] additional parameters for the dgn2shp() function.

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> carrs.dgn2shp()
To convert all available .dgn files of "CARRS" to shapefiles?
 [No]|Yes: yes
Converting "... ... _Overline_Bridge.dgn" at "data\CARRS\Overline bridges" ...␣
 ↪Done.
Converting "... ... _Underline_Bridge.dgn" at "data\CARRS\Underline bridges" ...
 ↪ Done.
Converting "... ... _Retaining_Wall.dgn" at "data\CARRS\Retaining walls" ...␣
 ↪Done.
Converting "... ... _TunnelPortal.dgn" at "data\CARRS\Tunnels" ... Done.
```

CARRS.import_overline_bridges_shp

CARRS.**import_overline_bridges_shp**(***kwargs*)

Import shapefile data (converted from the DGN data) of *overline bridges* into the project database.

Parameters

kwargs – Parameters of the method src.preprocessor.CARRS._import_dgn_shp().

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> carrs.import_overline_bridges_shp(if_exists='replace')
To import the DGN shapefile of overline bridges into "CARRS"."Overline_bridges"?
 [No]|Yes: yes
Importing the data ... Done.
```

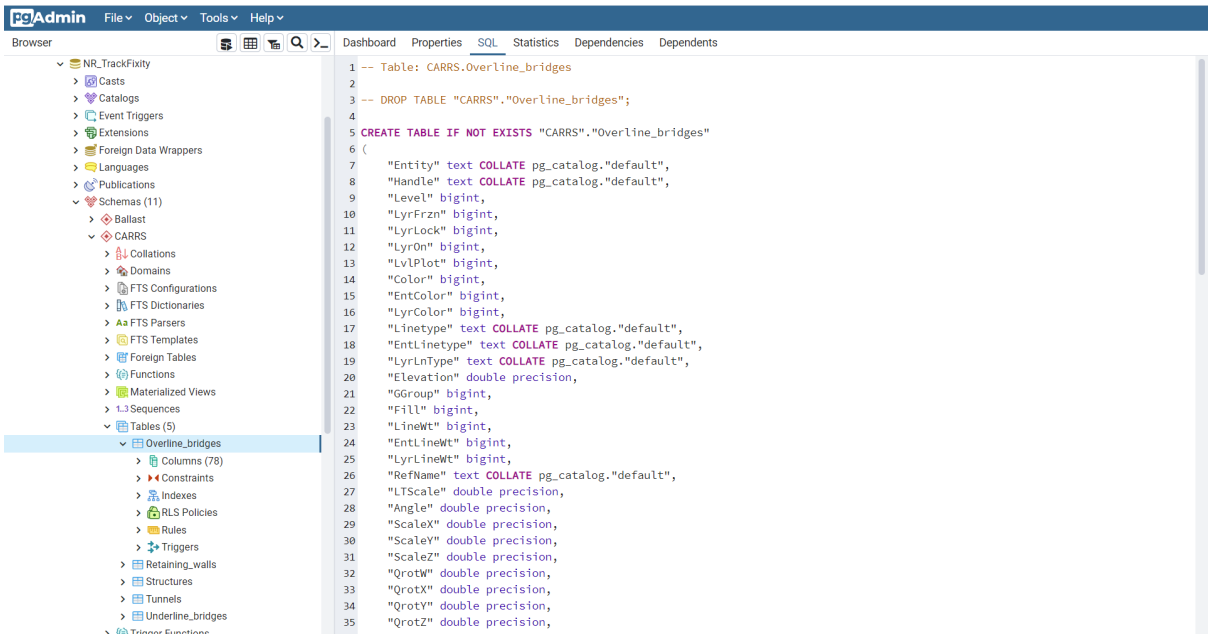


Figure 8: Snapshot of the "CARRS"."Overline_bridges" table.

CARRS.import_retaining_walls_shp

CARRS.**import_retaining_walls_shp**(***kwargs*)

Import shapefile data (converted from the DGN data) of *retaining walls* into the project database.

Parameters

kwargs – Parameters of the method `src.preprocessor.CARRS._import_dgn_shp()`.

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> carrs.import_retaining_walls_shp(if_exists='replace')
To import shapefile of retaining walls into the table "CARRS"."Retaining_walls"?
 [No]|Yes: yes
Importing the data ... Done.
```

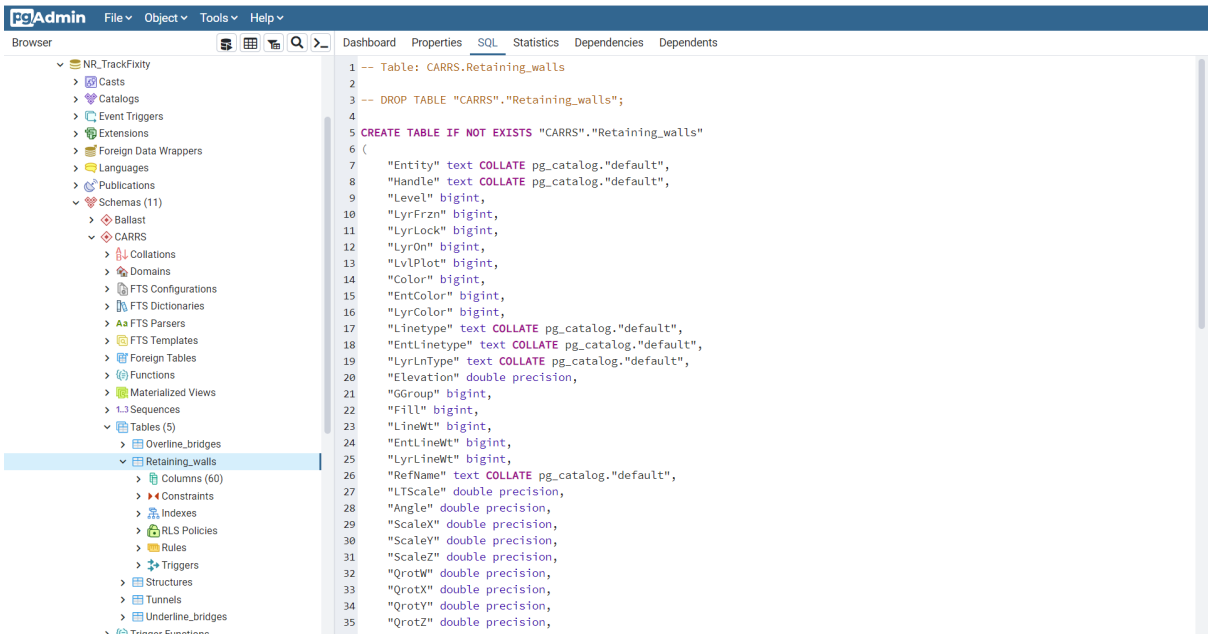


Figure 9: Snapshot of the "CARRS"."Retaining_walls" table.

CARRS.import_structures

CARRS.**import_structures**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import data of structures into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **confirmation_required** (*bool*) – Whether confirmation is required to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the pyhelpers.dbms.PostgreSQL.import_data method.

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> carrs.import_structures(if_exists='replace')
To import data of structures into the table "CARRS"."Structures"?
 [No]|Yes: yes
Importing the data ... Done.
```

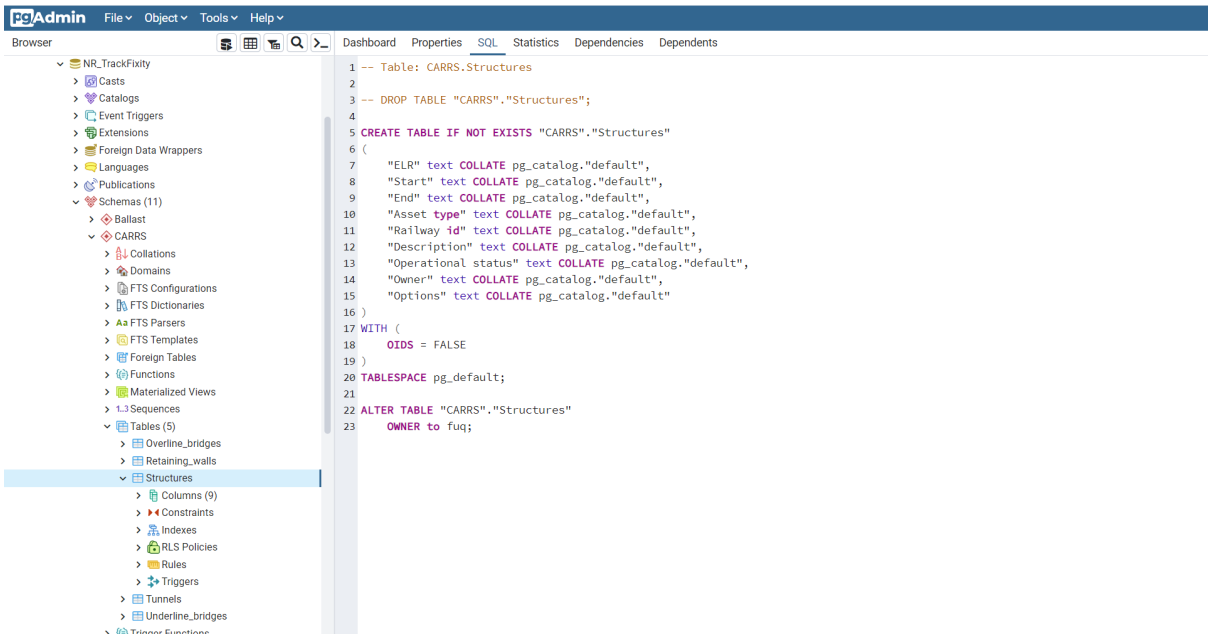


Figure 10: Snapshot of the "CARRS"."Structures" table.

CARRS.import_tunnels_shp

CARRS.**import_tunnels_shp**(***kwargs*)

Import shapefile data (converted from the DGN data) of *tunnels* into the project database.

Parameters

kwargs – Parameters of the method `src.preprocessor.CARRS._import_dgn_shp()`.

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> carrs.import_tunnels_shp(if_exists='replace')
To import shapefile of tunnels into the table "CARRS"."Tunnels"?
 [No]|Yes: yes
Importing the data ... Done.
```

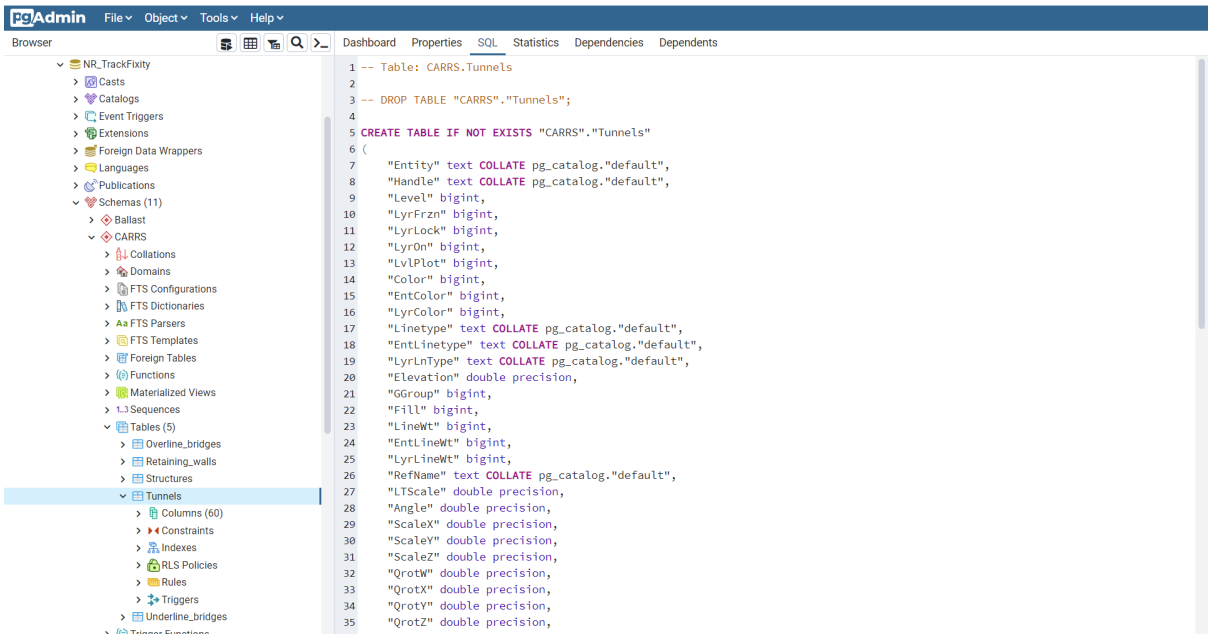


Figure 11: Snapshot of the "CARRS"."Tunnels" table.

CARRS.import_underline_bridges_shp

CARRS.**import_underline_bridges_shp**(***kwargs*)

Import shapefile data (converted from the DGN data) of *underline bridges* into the project database.

Parameters

kwargs – Parameters of the method `src.preprocessor.CARRS._import_dgn_shp()`.

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> carrs.import_underline_bridges_shp(if_exists='replace')
To import the DGN shapefile of underline bridges into "CARRS"."Underline_bridges
↪"?
 [No]|Yes: yes
Importing the data ... Done.
```

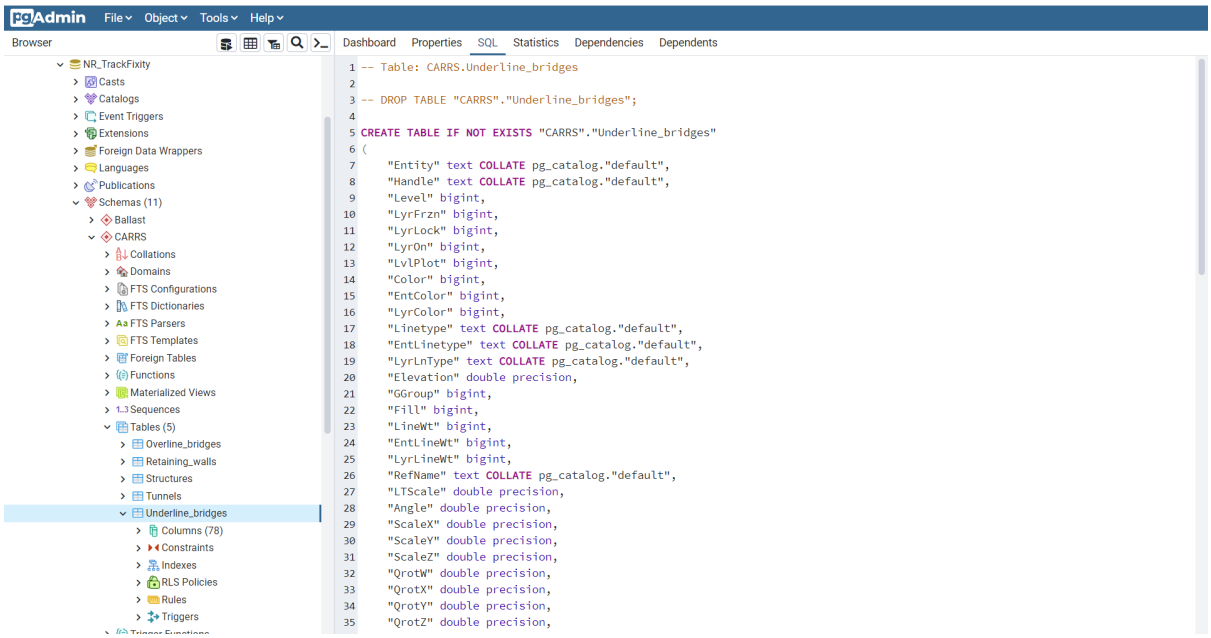


Figure 12: Snapshot of the "CARRS"."Underline_bridges" table.

CARRS.load_overline_bridges_shp

CARRS.**load_overline_bridges_shp**(***kwargs*)

Load the shapefile data (converted from the DGN data) of *overline bridges* from the project database.

Parameters

kwargs – Parameters of the method `src.preprocessor.carrs.CARRS._load_dgn_shp()`.

Returns

Data of the shapefile (converted from the DGN data) of *overline bridges*.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> ol_bdg_shp = carrs.load_overline_bridges_shp(elr=['ECM7', 'ECM8'])
>>> ol_bdg_shp.head()
    Entity  ...                                           geometry
0    Point  ...  POINT Z (423181.7536000004 610967.6328999996 0)
1    Point  ...  POINT Z (423785.0826000003 613768.3622999992 0)
2    Point  ...         POINT Z (423624.1886 616374.8772999998 0)
3    Point  ...         POINT Z (423184.8782000002 616951.3377 0)
4    Point  ...         POINT Z (421395.7363 620237.2636999991 0)
[5 rows x 78 columns]
```

CARRS.load_retaining_walls_shp

CARRS.**load_retaining_walls_shp**(***kwargs*)

Load the shapefile data (converted from the DGN data) of *retaining walls* from the project database.

Parameters
kwargs – Parameters of the method src.preprocessor.carrs.CARRS._load_dgn_shp().

Returns
Data of the shapefile (converted from the DGN data) of *retaining walls*.

Return type
pandas.DataFrame

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> retg_walls_shp = carrs.load_retaining_walls_shp(elr=['ECM7', 'ECM8'])
>>> retg_walls_shp.head()
    Entity  ...                                       geometry
0    Point  ...  POINT Z (425355.5612000003 564306.7263999991 0)
1    Point  ...  POINT Z (425448.4572000001 564368.4912999999 0)
2    Point  ...        POINT Z (425454.0697999997 564371.4035 0)
3    Point  ...  POINT Z (425467.7002999997 564378.4759999998 0)
4    Point  ...  POINT Z (425483.8436000003 564386.5826999992 0)
[5 rows x 60 columns]
```

CARRS.load_tunnels_shp

CARRS.**load_tunnels_shp**(***kwargs*)

Load the shapefile data (converted from the DGN data) of *tunnels* from the project database.

Parameters
kwargs – Parameters of the method src.preprocessor.carrs.CARRS._load_dgn_shp().

Returns
Data of the shapefile (converted from the DGN data) of *tunnels*.

Return type
pandas.DataFrame

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> tunl_shp = carrs.load_tunnels_shp(elr=['ECM7', 'ECM8'])
>>> tunl_shp.head()
```

(continues on next page)

(continued from previous page)

```
    Entity  ...                                          geometry
0    Point  ...  POINT Z (425355.5612000003 564306.7263999991 0)
1    Point  ...  POINT Z (425448.4572000001 564368.4912999999 0)
2    Point  ...        POINT Z (425454.0697999997 564371.4035 0)
3    Point  ...  POINT Z (425467.7002999997 564378.4759999998 0)
4    Point  ...  POINT Z (425483.8436000003 564386.5826999992 0)
[5 rows x 60 columns]
```

CARRS.load_underline_bridges_shp

CARRS.**load_underline_bridges_shp**(***kwargs*)

Load the shapefile data (converted from the DGN data) of *underline bridges* from the project database.

Parameters

kwargs – Parameters of the method `src.preprocessor.carrs.CARRS._load_dgn_shp()`.

Returns

Data of the shapefile (converted from the DGN data) of underline bridges.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> ul_bdg_shp = carrs.load_underline_bridges_shp(elr=['ECM7', 'ECM8'])
>>> ul_bdg_shp.head()
    Entity  ...                                          geometry
0    Point  ...  POINT Z (424644.8838999998 563787.7875999995 0)
1    Point  ...  POINT Z (424644.8838999998 563787.7875999995 0)
2    Point  ...  POINT Z (423963.1664000005 609154.5935999993 0)
3    Point  ...  POINT Z (423856.8586999997 609453.7774999999 0)
4    Point  ...  POINT Z (423721.8256000001 609828.4415000007 0)
[5 rows x 78 columns]
```

CARRS.map_view

CARRS.**map_view**(*structure_name*, *desc_col_name='DESCRIPTION'*, *sample=True*, *marker_colour='blue'*, *layer_name='Point'*, *update=False*, *verbose=True*)

Make a map view of a CARRS item.

Parameters

- **structure_name** (*str*) – Name of an item; options include `{'overline bridge', 'underline bridge', 'retaining wall', 'tunnel'}`.

- **desc_col_name** (*str*) – Name of a column that describes markers; defaults to `'DESCRIPTION'`.
- **sample** (*bool | int*) – Whether to draw a sample, or a given sample size; defaults to `True`.
- **marker_colour** (*str*) – Colour of markers; defaults to `'blue'`.
- **layer_name** (*str*) – Name of a layer; defaults to `'Point'`.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `True`.

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
```

Overline bridges:

```
>>> # # 100 random examples:
>>> carrs.map_view('Overline bridges', sample=100, marker_colour='blue')
```

Figure 13: Examples of overline bridges.

Underline bridges:

```
>>> # 100 random examples:
>>> carrs.map_view('Underline bridges', sample=100, marker_colour='orange')
```

Figure 14: Examples of underline bridges.

Retaining walls:

```
>>> # 100 random examples:
>>> carrs.map_view('Retaining walls', 'ASSET_DESCRIPTION', 100, marker_colour=
 ↪'green')
```

Figure 15: Random examples of retaining walls.

Tunnels:

```
>>> # 100 random examples:
>>> carrs.map_view('Tunnels', sample=100, marker_colour='lightred')
```

Figure 16: Random examples of tunnels.

CARRS.read_overline_bridges_shp

CARRS.**read_overline_bridges_shp**(***kwargs*)

Read the shapefile (converted from the DGN data) of *overline bridges* from a local directory.

Parameters

kwargs – Parameters of the method `src.preprocessor.carrs.CARRS._read_dgn_shp()`.

Returns

Data of the shapefile of *overline bridges*.

Return type

dict | None

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> ol_bdg_shp = carrs.read_overline_bridges_shp()
>>> type(ol_bdg_shp)
dict
>>> list(ol_bdg_shp.keys())
['Annotation', 'MultiPatch', 'Point', 'Polygon', 'Polyline']
>>> ol_bdg_shp['MultiPatch'].empty
True
>>> ol_bdg_shp['Annotation'].empty
True
>>> ol_bdg_shp['Point'].empty
False
>>> ol_bdg_shp['Polygon'].empty
True
>>> ol_bdg_shp['Polyline'].empty
True
>>> ol_bdg_shp['Point'].head()
      Entity  ...                                geometry
0      Point  ...   POINT Z (150671.813 31303.480 0.000)
1      Point  ...   POINT Z (151667.807 31945.272 0.000)
2      Point  ...   POINT Z (152952.274 33955.715 0.000)
3      Point  ...   POINT Z (156042.572 37403.277 0.000)
4      Point  ...   POINT Z (158027.581 38004.746 0.000)
[5 rows x 78 columns]
```

CARRS.read_prj_metadata

CARRS.**read_prj_metadata**(*update=False*, *parser='osr'*, *as_dict=True*, *verbose=True*)

Read metadata of projection for the DGN files/shapefiles.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **parser** (*str*) – Name of package used for reading the PRJ file; options include `{'osr', 'pycrs'}`; defaults to `'osr'`.
- **as_dict** (*bool*) – Whether to return the data as a dictionary; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `True`.

Returns

Metadata of the projection for the DGN files/shapefiles.

Return type

dict | None

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
```

(continues on next page)

(continued from previous page)

```
>>> dgn_shp_prj = carrs.read_prj_metadata()
>>> type(dgn_shp_prj)
dict
>>> list(dgn_shp_prj.keys())
['PROJCS', 'Shapefile']
>>> type(dgn_shp_prj['PROJCS'])
dict
>>> list(dgn_shp_prj['PROJCS'].keys())
['proj', 'lat_0', 'lon_0', 'k', 'x_0', 'y_0', 'ellps', 'units']
>>> dgn_shp_prj['Shapefile']
  ORDER_ID  ...                                           geometry
0    69433  ...  POLYGON ((307732.043 1047233.175, 702872.281 4...
[1 rows x 4 columns]
```

CARRS.read_retaining_walls_shp

CARRS.**read_retaining_walls_shp**(***kwargs*)

Read the shapefile (converted from the DGN data) of *retaining walls* from a local directory.

Parameters

kwargs – Parameters of the method `src.preprocessor.carrs.CARRS._read_dgn_shp()`.

Returns

Data of the shapefile of *retaining walls*.

Return type

dict | None

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> retg_walls_shp = carrs.read_retaining_walls_shp()
>>> type(retg_walls_shp)
dict
>>> list(retg_walls_shp.keys())
['Annotation', 'MultiPatch', 'Point', 'Polygon', 'Polyline']
>>> retg_walls_shp['Annotation'].empty
True
>>> retg_walls_shp['MultiPatch'].empty
True
>>> retg_walls_shp['Point'].empty
False
>>> retg_walls_shp['Polygon'].empty
True
>>> retg_walls_shp['Polyline'].empty
True
>>> retg_walls_shp['Point'].head()
    Entity  ...                                geometry
0    Point  ...  POINT Z (152580.106 33145.415 0.000)
1    Point  ...  POINT Z (153723.038 35252.128 0.000)
2    Point  ...  POINT Z (152157.389 39942.775 0.000)
```

(continues on next page)

(continued from previous page)

```
3      Point  ...   POINT Z (152344.545 39897.870 0.000)
4      Point  ...   POINT Z (152531.952 39017.593 0.000)
[5 rows x 60 columns]
```

CARRS.read_structures

CARRS.**read_structures**(*update=False*, *verbose=False*)

Read data of structures from a local directory.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Returns

Data of structures from *CARRS*.

Return type

pandas.DataFrame | None

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> # structures_data = carrs.read_structures(update=True, verbose=True)
>>> # Parsing data of structures ... Done.
>>> # Updating "Structures.pkl" at "data\CARRS\Structures" ... Done.
>>> data = carrs.read_structures()
>>> data
      ELR    Start End  ... Operational status                    Owner Options
0    ECM8  10.0004      ...        Functionary             Outside Party
1    ECM8  10.0266      ...        Functionary  Network Rail (CE-Struct)
2    ECM8  10.0308      ...        Functionary  Network Rail (CE-Struct)
3    ECM8  10.0324      ...        Functionary  Network Rail (CE-Struct)
4    ECM8  10.0330      ...        Functionary  Network Rail (CE-Struct)
..    ...      ...  ..  ...                ...                        ...      ...
195  ECM8  53.0880      ...        Functionary  Network Rail (CE-Struct)
196  ECM8  53.1320      ...        Functionary  Network Rail (CE-Struct)
197  ECM8  53.1673      ...        Functionary            NR Maintenance
198  ECM8  54.0264      ...        Functionary  Network Rail (CE-Struct)
199  ECM8  54.0286      ...        Functionary  Network Rail (CE-Struct)
[200 rows x 9 columns]
```

CARRS.read_tunnels_shp

CARRS.**read_tunnels_shp**(***kwargs*)

Read the shapefile (converted from the DGN data) of *tunnels* from a local directory.

Parameters

kwargs – Parameters of the method `src.preprocessor.carrs.CARRS._read_dgn_shp()`.

Returns

Data of the shapefile (converted from the DGN data) of *tunnels*.

Return type

dict | None

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> tunl_shp = carrs.read_tunnels_shp()
>>> type(tunl_shp)
dict
>>> list(tunl_shp.keys())
['Annotation', 'MultiPatch', 'Point', 'Polygon', 'Polyline']
>>> tunl_shp['Annotation'].empty
True
>>> tunl_shp['MultiPatch'].empty
True
>>> tunl_shp['Point'].empty
False
>>> tunl_shp['Polygon'].empty
True
>>> tunl_shp['Polyline'].empty
True
>>> tunl_shp['Point'].head()
     Entity  ...                                   geometry
0     Point  ...  POINT Z (175570.798 782711.751 0.000)
1     Point  ...  POINT Z (185539.688 781374.833 0.000)
2     Point  ...  POINT Z (191942.787 780080.803 0.000)
3     Point  ...  POINT Z (221042.577 654746.431 0.000)
4     Point  ...  POINT Z (227213.665 676197.641 0.000)
[5 rows x 69 columns]
```

CARRS.read_underline_bridges_shp

CARRS.**read_underline_bridges_shp**(***kwargs*)

Read the shapefile (converted from the DGN data) of *underline bridges* from a local directory.

Parameters

kwargs – Parameters of the method `src.preprocessor.carrs.CARRS._read_dgn_shp()`.

Returns
Data of the shapefile of *underline bridges*.

Return type
dict | None

Examples:

```
>>> from src.preprocessor import CARRS
>>> carrs = CARRS()
>>> ul_bdg_shp = carrs.read_underline_bridges_shp()
>>> type(ul_bdg_shp)
dict
>>> list(ul_bdg_shp.keys())
['Annotation', 'MultiPatch', 'Point', 'Polygon', 'Polyline']
>>> ul_bdg_shp['Annotation'].empty
True
>>> ul_bdg_shp['MultiPatch'].empty
True
>>> ul_bdg_shp['Point'].empty
False
>>> ul_bdg_shp['Polygon'].empty
True
>>> ul_bdg_shp['Polyline'].empty
True
>>> ul_bdg_shp['Point'].head()
      Entity  ...                                geometry
0      Point  ...   POINT Z (153777.071 35335.486 0.000)
1      Point  ...   POINT Z (154081.346 35669.935 0.000)
2      Point  ...   POINT Z (162641.973 39207.381 0.000)
3      Point  ...   POINT Z (166286.374 40370.296 0.000)
4      Point  ...   POINT Z (166318.680 40394.695 0.000)
[5 rows x 78 columns]
```

2.2.3 CNM

class src.preprocessor.**CNM**(*db_instance=None*)

Corporate Network Model.

The model is Network Rail's geospatial information system.

Note: This class currently handles only DGN data of waymarks.

Parameters
db_instance (TrackFixityDB | *None*) – PostgreSQL database instance; defaults to None.

Variables

- **wm_dgn_pathnames** (*list*) – List of paths to the original DGN data files.

- **wm_shp_pathnames** (*list*) – List of lists, each containing paths to shapefiles converted from DGN.
- **wm_item_names** (*list*) – List of names of items available in this data category.
- **db_instance** (TrackFixityDB) – PostgreSQL database instance.

Examples:

```
>>> from src.preprocessor import CNM
>>> cnm = CNM()
>>> cnm.NAME
'Corporate Network Model'
```

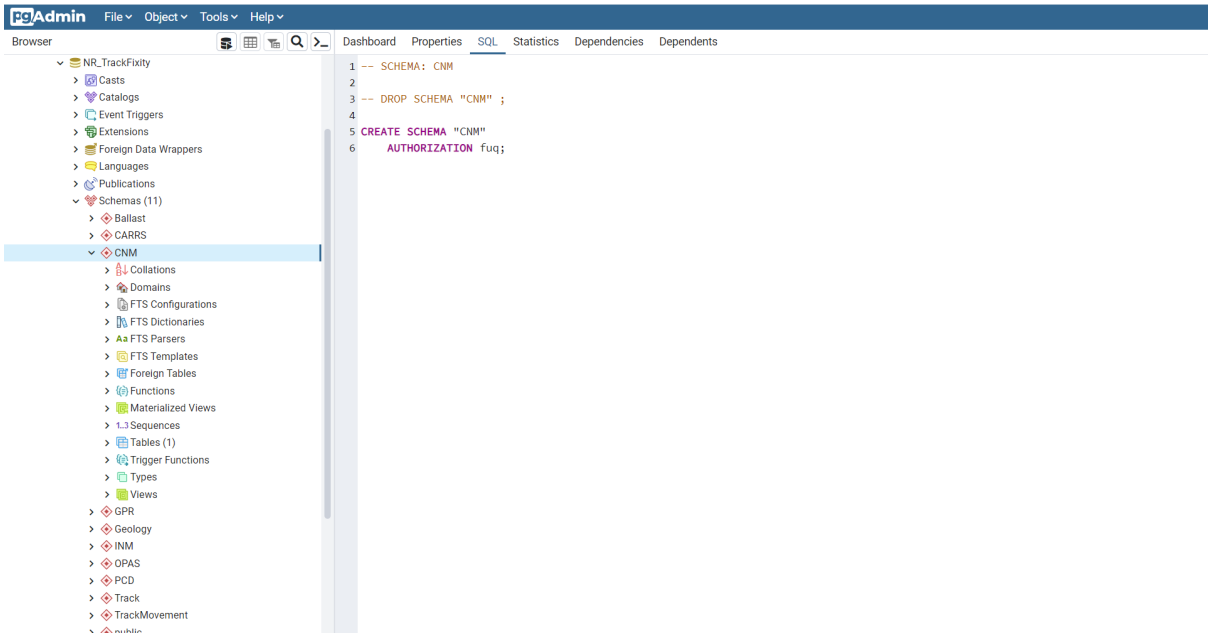


Figure 17: Snapshot of the *CNM* schema.

Attributes:

ACRONYM	Acronym for the data name.
DATA_DIR	Pathname of a local directory where the CNM data is stored.
NAME	Data name.
PROJ_DIRNAME	Directory name of projection/boundary shapefiles.
PROJ_FILENAME	Name of the projection file.
SCHEMA_NAME	Name of the schema for storing the CNM data.
WM_DIRNAME	Directory name of *waymarks* data.
WM_TABLE_NAME	Name of the table for storing the *waymarks* data.

CNM.ACRONYM

CNM.**ACRONYM: str = 'CNM'**

Acronym for the data name.

CNM.DATA_DIR

CNM.**DATA_DIR: str = 'data\\CNM'**

Pathname of a local directory where the CNM data is stored.

CNM.NAME

CNM.**NAME: str = 'Corporate Network Model'**

Data name.

CNM.PROJ_DIRNAME

CNM.**PROJ_DIRNAME: str = 'Projection'**

Directory name of projection/boundary shapefiles.

CNM.PROJ_FILENAME

CNM.**PROJ_FILENAME: str = 'Order_69705_Polygon'**

Name of the projection file.

CNM.SCHEMA_NAME

CNM.**SCHEMA_NAME: str = 'CNM'**

Name of the schema for storing the CNM data.

CNM.WM_DIRNAME

CNM.**WM_DIRNAME: str = 'Waymarks'**

Directory name of *waymarks* data.

CNM.WM_TABLE_NAME

CNM.**WM_TABLE_NAME: str = 'Waymarks'**

Name of the table for storing the *waymarks* data.

Methods:

dgn2shp([dat_name, confirmation_required, ...])	Convert DGN files of CNM data to shapefiles.
import_waymarks_shp([update, ...])	Import shapefile data (converted from the DGN data) of waymarks into the project database.
load_waymarks_shp([elr, column_names, ...])	Load the points layer of the waymarks shapefile (for a given ELR or ELRs) from the project database.
make_pseudo_waymarks(waymarks)	Make pseudo mileages and geometry objects of waymarks.
map_view([item_name, layer_name, ...])	Make a map view of a given item.
read_prj_metadata([update, parser, as_dict, ...])	Read metadata of projection for the DGN files/shapefiles from a local directory.
read_waymarks_shp([update, verbose])	Read shapefile data (converted from the DGN data) of waymarks from a local directory.

CNM.dgn2shp

CNM.**dgn2shp**(*dat_name='Waymarks'*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Convert DGN files of CNM data to shapefiles.

Parameters

- **dat_name** (*str | None*) – Name of the data; defaults to `'Waymarks'`.
- **confirmation_required** (*bool*) – Whether confirmation is required to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the function `dgn2shp()`.

Examples:

```
>>> from src.preprocessor import CNM
>>> cnm = CNM()
```

(continues on next page)

(continued from previous page)

```
>>> cnm.dgn2shp()
To convert .dgn files of "Waymarks" to shapefiles?
 [No]|Yes: yes
Converting "CNM_ADMIN.NetworkWaymarks.dgn" at "data\CNM\Waymarks" ... Done.
```

CNM.import_waymarks_shp

CNM.**import_waymarks_shp**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import shapefile data (converted from the DGN data) of waymarks into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **confirmation_required** (*bool*) – Whether confirmation is required to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import CNM
>>> cnm = CNM()
>>> cnm.import_waymarks_shp(if_exists='replace')
To import shapefile of waymarks into the table "CNM"."Waymarks"?
 [No]|Yes: yes
Importing the data ... Done.
```

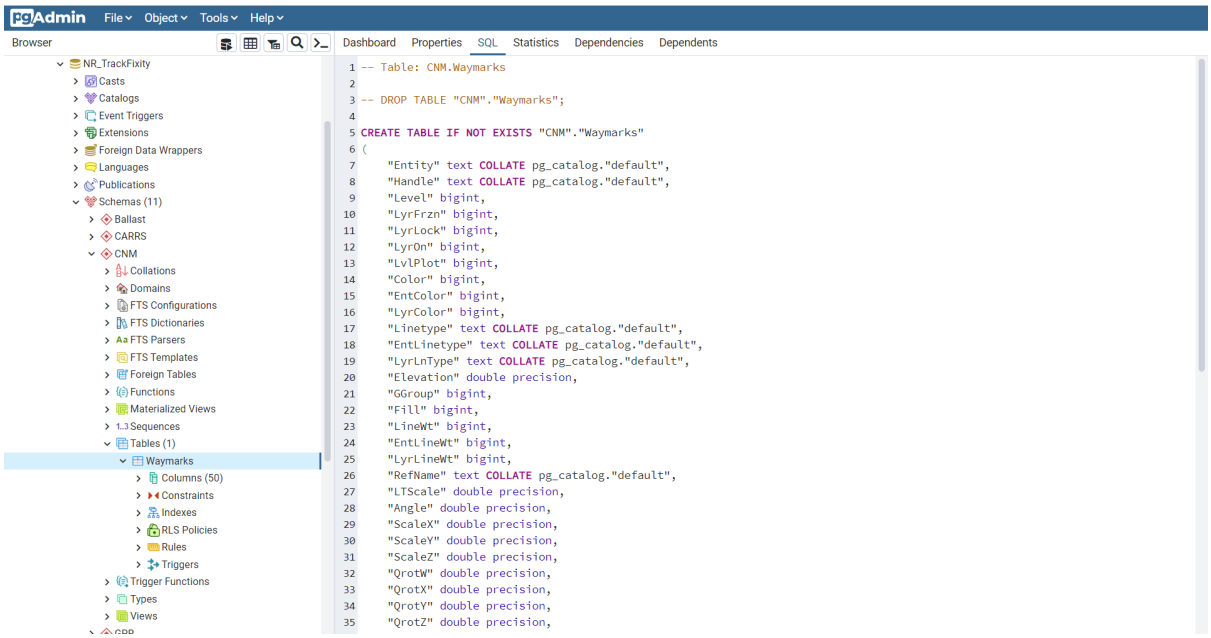


Figure 18: Snapshot of the "CNM"."Waymarks" table.

CNM.load_waymarks_shp

CNM.**load_waymarks_shp**(*elr=None*, *column_names=None*, *fmt_nr_mileage=False*, ***kwargs*)

Load the points layer of the waymarks shapefile (for a given ELR or ELRs) from the project database.

Parameters

- **elr** (*str | list | tuple | None*) – Engineer's Line Reference(s); defaults to `None`.
- **column_names** (*str | list | None*) – Names of (a subset of) columns to be queried; defaults to `None`.
- **fmt_nr_mileage** (*bool*) – Whether to add formatted Network Rail mileage data; defaults to `False`.
- **kwargs** – [Optional] parameters for the method pyhelpers.dbms.PostgreSQL.read_sql_query.

Returns

Data of the waymarks shapefile.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import CNM
>>> cnm = CNM()
```

(continues on next page)

(continued from previous page)

```
>>> waymarks_points = cnm.load_waymarks_shp()
>>> waymarks_points.head()
      Entity  ...                                           geometry
0      Point  ...  POINT Z (492436.76070000045 168198.35700000077 0)
1      Point  ...   POINT Z (491806.33029999956 168202.1041000001 0)
2      Point  ...  POINT Z (491737.74010000005 167755.89090000093 0)
3      Point  ...  POINT Z (491787.06960000005 167357.06870000064 0)
4      Point  ...   POINT Z (491888.97809999995 166960.9595999997 0)
[5 rows x 50 columns]
>>> waymarks_points = cnm.load_waymarks_shp(elr='ECM8')
>>> waymarks_points.head()
    Entity  ...                                           geometry
0    Point  ...  POINT Z (327289.9845000003 674251.0335000008 0)
1    Point  ...  POINT Z (327696.2715999996 674194.5047999993 0)
2    Point  ...  POINT Z (328097.8353000004 674240.6223000009 0)
3    Point  ...  POINT Z (328494.3197999997 674324.2128999997 0)
4    Point  ...  POINT Z (328893.0425000004 674278.1779999994 0)
[5 rows x 51 columns]
>>> waymarks_points = cnm.load_waymarks_shp(['ECM7', 'ECM8'], column_names=
↪'essential')
>>> waymarks_points.head()
     ELR  WAYMARK_VALUE                                       geometry
0   ECM7         0.0000  POINT Z (424615.7734000003 563820.6427999996 0)
1   ECM7         0.0440  POINT Z (425015.6885000002 563885.4956999999 0)
2   ECM7         0.0880  POINT Z (425266.2346999999 564189.9217000008 0)
3   ECM7         0.1320  POINT Z (425568.8497000001 564439.0111999996 0)
4   ECM7         1.0000  POINT Z (425931.0060999999 564623.0390000008 0)
[5 rows x 3 columns]
```

CNM.make_pseudo_waymarks

static CNM.**make_pseudo_waymarks**(*waymarks*)

Make pseudo mileages and geometry objects of waymarks.

Parameters

waymarks (*pandas.DataFrame*) – Data of waymarks.

Returns

Data of waymarks with pseudo mileages.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import CNM
>>> cnm = CNM()
>>> waymarks = cnm.load_waymarks_shp(['ECM7', 'ECM8'], column_names='essential')
>>> waymarks_ = cnm.make_pseudo_waymarks(waymarks)
>>> waymarks_.head()
     ELR  ...                                    pseudo_geometry
0   ECM7  ...  LINESTRING Z (424615.7734000003 563820.6427999...
1   ECM7  ...  LINESTRING Z (425015.68850000016 563885.495699...
2   ECM7  ...  LINESTRING Z (425266.2346999999 564189.9217000...
```

(continues on next page)

(continued from previous page)

```
3    ECM7  ...  LINESTRING Z (425568.84970000014 564439.011199...
4    ECM7  ...  LINESTRING Z (425931.0060999999 564623.0390000...
[5 rows x 6 columns]
```

CNM.map_view

CNM.**map_view**(*item_name='Waymarks'*, *layer_name='Point'*, *desc_col_name='ASSETID'*, *sample=True*, *marker_colour='purple'*, *update=False*, *verbose=True*)

Make a map view of a given item.

Parameters

- **item_name** (*str*) – Name of the item; defaults to `'Waymarks'`.
- **layer_name** (*str*) – Name of the layer; defaults to `'Point'`.
- **desc_col_name** (*str*) – Name of the column that describes markers; defaults to `'ASSETID'`.
- **sample** (*bool | int*) – Whether to draw a sample or a specific sample size; defaults to `True`.
- **marker_colour** (*str*) – Colour of markers; defaults to `'purple'`.
- **update** (*bool*) – Whether to reprocess the original data files; defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `True`.

Examples:

```
>>> from src.preprocessor import CNM
>>> cnm = CNM()
>>> cnm.map_view(desc_col_name='ASSETID', sample=100)
```

Figure 19: Examples of waymarks.

CNM.read_prj_metadata

CNM.**read_prj_metadata**(*update=False*, *parser='osr'*, *as_dict=True*, *verbose=False*)

Read metadata of projection for the DGN files/shapefiles from a local directory.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **parser** (*str*) – Name of the package used to read the PRJ file; options include `{'osr', 'pycrs'}`; defaults to `'osr'`.
- **as_dict** (*bool*) – Whether to return the data as a dictionary; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in

the console; defaults to `False`.

Returns
Metadata of the projection for the DGN files/shapefiles.

Return type
dict | None

Examples:

```
>>> from src.preprocessor import CNM
>>> cnm = CNM()
>>> dgn_shp_prj = cnm.read_prj_metadata()
>>> type(dgn_shp_prj)
dict
>>> list(dgn_shp_prj.keys())
['PROJCS', 'Shapefile']
>>> list(dgn_shp_prj['PROJCS'].keys())
['proj', 'lat_0', 'lon_0', 'k', 'x_0', 'y_0', 'ellps', 'units']
>>> dgn_shp_prj['Shapefile']
  ORDER_ID  ...                                           geometry
0    69705  ...  POLYGON ((257789.191 1047540.101, 473308.767 1...
[1 rows x 4 columns]
```

CNM.read_waymarks_shp

CNM.**read_waymarks_shp**(*update=False*, *verbose=False*)

Read shapefile data (converted from the DGN data) of waymarks from a local directory.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Returns
DGN-converted shapefile data of waymarks.

Return type
dict

Examples:

```
>>> from src.preprocessor import CNM
>>> cnm = CNM()
>>> waymarks_shp = cnm.read_waymarks_shp()
>>> type(waymarks_shp)
dict
>>> list(waymarks_shp.keys())
['Annotation', 'MultiPatch', 'Point', 'Polygon', 'Polyline']
>>> waymarks_shp['Annotation'].empty
True
```

(continues on next page)

(continued from previous page)

```
>>> waymarks_shp['MultiPatch'].empty
True
>>> waymarks_shp['Point'].empty
False
>>> waymarks_shp['Polygon'].empty
True
>>> waymarks_shp['Polyline'].empty
True
>>> waymarks_shp['Point'].head()
      Entity  ...                                geometry
0      Point  ...   POINT Z (150011.002 31282.015 0.000)
1      Point  ...   POINT Z (150804.322 31339.413 0.000)
2      Point  ...   POINT Z (154117.981 35706.001 0.000)
3      Point  ...   POINT Z (152838.030 38773.046 0.000)
4      Point  ...   POINT Z (155501.881 37036.022 0.000)
[5 rows x 50 columns]
```

2.2.4 Geology

class src.preprocessor.**Geology**(*db_instance=None*)

Geology.

Note: This class currently handles only a summary about the geology.

Parameters

db_instance (TrackFixityDB / *None*) – PostgreSQL database instance; defaults to None.

Variables

- **dtypes** (*dict*) – Field types of the geology data.
- **db_instance** (TrackFixityDB) – PostgreSQL database instance used for operations.

Examples:

```
>>> from src.preprocessor import Geology
>>> geol = Geology()
>>> geol.NAME
'Geology'
```

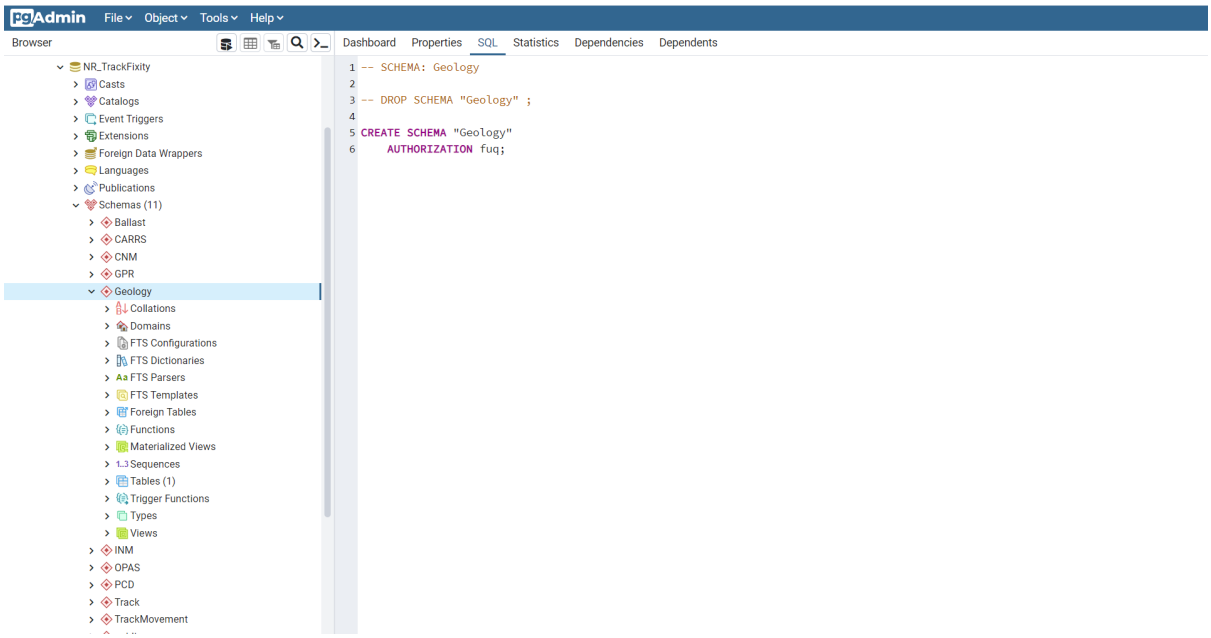


Figure 20: Snapshot of the *Geology* schema.

Attributes:

DATA_DIR	Pathname of a local directory where the geology data is stored.
GEOL_FILENAME	Filename of the original geology data file.
GEOL_TABLE_NAME	Name of the table for storing the geology data.
NAME	Data name.
SCHEMA_NAME	Name of schema for storing the geology data.

Geology.DATA_DIR

`Geology.DATA_DIR: str = 'data\\Geology'`

Pathname of a local directory where the geology data is stored.

Geology.GEOL_FILENAME

Geology.**GEOL_FILENAME: str = 'Geology'**

Filename of the original geology data file.

Geology.GEOL_TABLE_NAME

Geology.**GEOL_TABLE_NAME: str = 'Geology'**

Name of the table for storing the geology data.

Geology.NAME

Geology.**NAME: str = 'Geology'**

Data name.

Geology.SCHEMA_NAME

Geology.**SCHEMA_NAME: str = 'Geology'**

Name of schema for storing the geology data.

Methods:

import_summary([update, ...])	Import geology data into the project database.
load_summary([elr])	Load geology data from the project database.
read_summary([update, verbose])	Read geology data from a local directory.

Geology.import_summary

Geology.**import_summary**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import geology data into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **confirmation_required** (*bool*) – Whether confirmation is required to proceed; defaults to True.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to True.

- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import Geology
>>> geol = Geology()
>>> geol.import_summary(if_exists='replace')
To import data of geology into the table "Geology"."Geology"?
 [No]|Yes: yes
Importing the data ... Done.
```

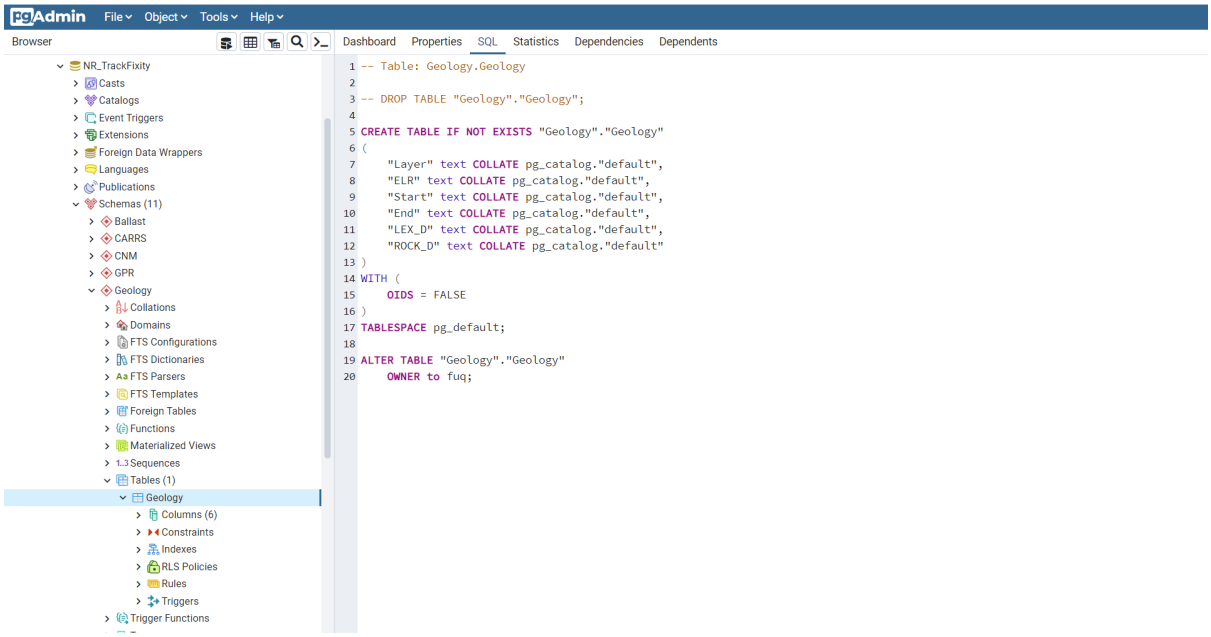


Figure 21: Snapshot of the "Geology"."Geology" table.

Geology.load_summary

Geology.**load_summary**(*elr=None*, ***kwargs*)

Load geology data from the project database.

Parameters

- **elr** (*str | list | tuple | None*) – Engineer's Line Reference(s); defaults to None.
- **kwargs** – [Optional] parameters for the method pyhelpers.dbms.PostgreSQL.read_sql_query.

Returns

Data of geology.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Geology
>>> geol = Geology()
>>> data = geol.load_summary(elr=['ECM7', 'ECM8'])
>>> data.shape
(662, 6)
```

Geology.read_summary

Geology.**read_summary**(*update=False*, *verbose=False*)

Read geology data from a local directory.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to False.

Returns

Data of geology.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Geology
>>> geol = Geology()
>>> data = geol.read_summary()
>>> data.shape
(55642, 6)
```

2.2.5 GPR

class src.preprocessor.**GPR**(*db_instance=None*)

Ground Penetrating Radar.

Note:

- This class handles GPR data, formatted as a bundle of three types of data files with the extensions ".DZX", ".DZG", and ".DZT".
- GPR data has not yet been considered in the *shaft* module for developing the current data model for this project. However, it may potentially be useful and could be included for further development of the data model and the algorithm for learning about track fixity.

Parameters

db_instance (TrackFixityDB | *None*) – PostgreSQL database instance; defaults to None.

Variables

- **data_dates** (*list*) – Date range of the original GPR data.
- **dtypes** (*dict*) – Field types of the GPR data.
- **db_instance** (TrackFixityDB) – PostgreSQL database instance used for operations.
- **schema_name** (*Callable*) – Name of the schema for storing the GPR data.

Examples:

```
>>> from src.preprocessor import GPR
>>> gpr = GPR()
>>> gpr.NAME
'Ground Penetrating Radar'
>>> gpr.data_dates  # '20200312 PM' -> '20200312'
['20200107', '20200110', '20200116', '20200312', '20200428', '20200531']
```

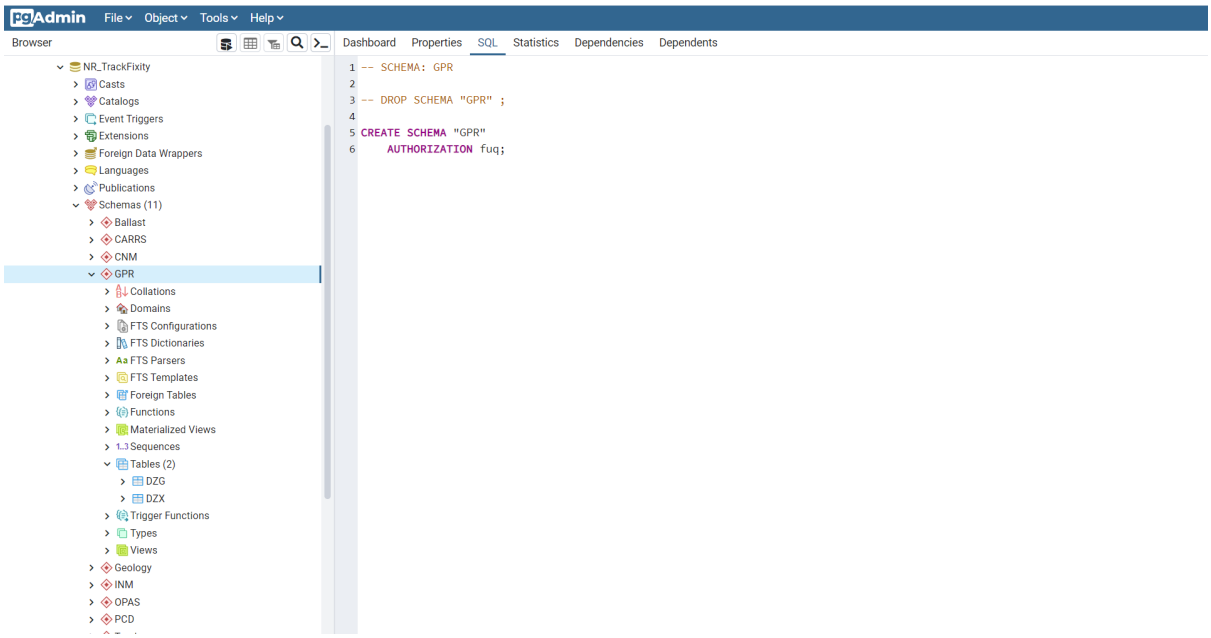


Figure 22: Snapshot of the *GPR* schema.

Attributes:

ACRONYM	Acronym for the data name.
DATA_DIR	Pathname of a local directory where the GPR data is stored.
DZG_TABLE_NAME	Name of the table for storing *DZG* data.
DZT_TABLE_NAME	Name of the table for storing *DZT* data.
DZX_TABLE_NAME	Name of the table for storing *DZX* data.
FILE_DATA_DIR	Pathname of a local directory where the GPR data files are stored.
NAME	Data name.
SCHEMA_NAME	Name of the schema for storing the GPR data.
ZIPFILE_DATA_DIR	Pathname of a local directory where zipped GPR data is stored for backup.

GPR.ACRONYM

GPR.**ACRONYM: str = 'GPR'**

Acronym for the data name.

GPR.DATA_DIR

GPR.**DATA_DIR: str = 'data\\GPR'**

Pathname of a local directory where the GPR data is stored.

GPR.DZG_TABLE_NAME

GPR.**DZG_TABLE_NAME: str = 'DZG'**

Name of the table for storing *DZG* data.

GPR.DZT_TABLE_NAME

GPR.**DZT_TABLE_NAME: str = 'DZT'**

Name of the table for storing *DZT* data.

GPR.DZX_TABLE_NAME

GPR.**DZX_TABLE_NAME: str = 'DZX'**

Name of the table for storing *DZX* data.

GPR.FILE_DATA_DIR

GPR.**FILE_DATA_DIR: str = 'data\\GPR\\files'**

Pathname of a local directory where the GPR data files are stored.

GPR.NAME

GPR.**NAME: str = 'Ground Penetrating Radar'**

Data name.

GPR.SCHEMA_NAME

GPR.**SCHEMA_NAME: str = 'GPR'**

Name of the schema for storing the GPR data.

GPR.ZIPFILE_DATA_DIR

GPR.**ZIPFILE_DATA_DIR: str = 'data_backups\\GPR\\files_zipped'**

Pathname of a local directory where zipped GPR data is stored for backup.

Methods:

get_files_info(gpr_date)	Get useful information about GPR data files.
import_dzg([update, confirmation_required, ...])	Import the DZG data into the project database.
*import_dzt_*([update, confirmation_required, ...])	TODO: Import the DZT data into the project database.
import_dzx([update, confirmation_required, ...])	Import the DZX data into the project database.
parse_dzg(path_to_dzg)	TODO: Parse a DZG data file.
*parse_dzt_*(path_to_dzt)	TODO: Parse a DZT data file.
parse_dzx(path_to_dzx[, head_tag])	Parse a DZX file (.DZX).
parse_gpr_log(path_to_gpr_log[, head_tag])	Parse a GPR log file (.lxml).
read_dzg([update, verbose])	Read DZG data.
read_dzg_by_date(gpr_date[, update, verbose])	Read DZG data for a given date (i.e. data folder).
read_dzt_by_date(gpr_date[, update, ...])	Read DZT data for a given date (i.e. data folder).
read_dzx([update, verbose])	Read DZX data.
read_dzx_by_date(gpr_date[, update, verbose])	Read DZX data for a given date (i.e. data folder).
unzip_backup_data(gpr_date[, ...])	Unzip GPR data files (up to the given date) in a local directory.
unzip_data([confirmation_required, verbose])	Unzip all available GPR data files from the backup in a local directory.
*visualise_dzt_*(path_to_dzt[, channel])	TODO: Visualise DZT data.

GPR.get_files_info

GPR.**get_files_info**(*gpr_date*)

Get useful information about GPR data files.

Parameters
gpr_date (*str*) – Date of the data (i.e. name of the data folder).

Returns
Useful information about GPR data files.

Return type
dict

Examples:

```
>>> from src.preprocessor import GPR
>>> gpr = GPR()
```

(continues on next page)

(continued from previous page)

```
>>> # gpr.unzip_backup_data(gpr_date='20200531', verbose=True)
>>> files_information = gpr.get_files_info(gpr_date='20200531')
>>> type(files_information)
dict
>>> len(files_information)
5
```

GPR.import_dzg

GPR.**import_dzg**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import the DZG data into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **confirmation_required** (*bool*) – Whether confirmation is required to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import GPR
>>> gpr = GPR()
>>> gpr.import_dzg()
To import DZG data into the table "GPR"."DZG"?
 [No]|Yes: yes
Importing the data ... Done.
```

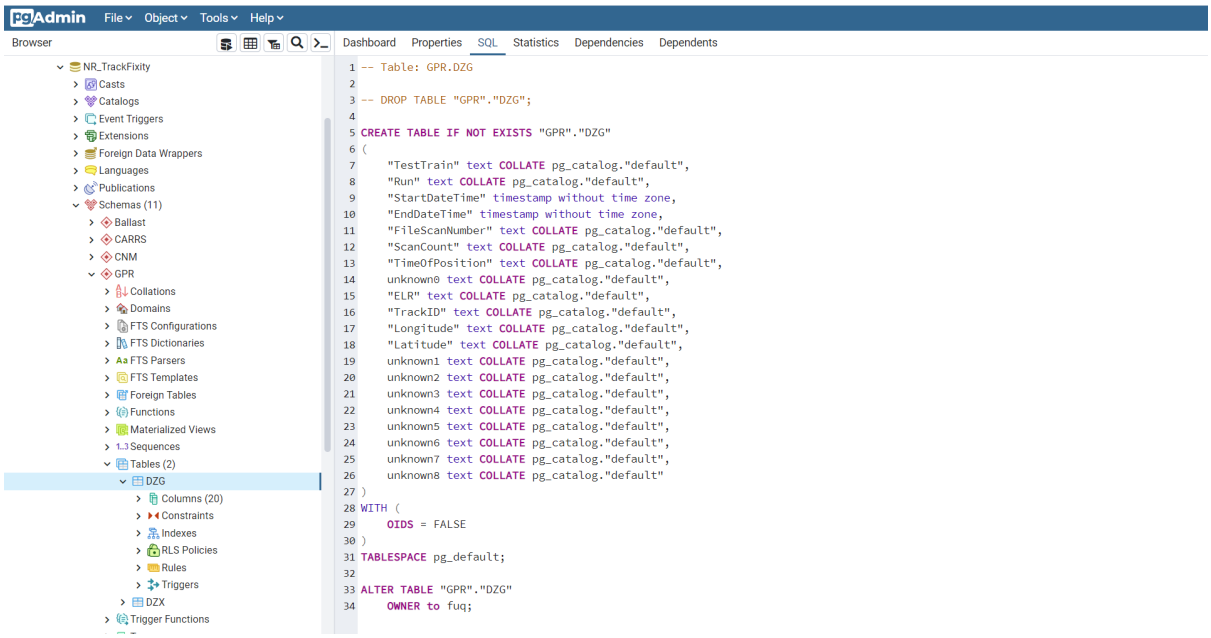


Figure 23: Snapshot of the "GPR"."DZG" table.

GPR.import_dzt_

GPR.**import_dzt_**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

TODO: Import the DZT data into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **confirmation_required** (*bool*) – Whether a confirmation is required to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

GPR.import_dzx

GPR.**import_dzx**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import the DZX data into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **confirmation_required** (*bool*) – Whether confirmation is required to proceed; defaults to `True`.

- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import GPR
>>> gpr = GPR()
>>> gpr.import_dzx(if_exists='replace')
To import DZX data into the table "GPR"."DZX"?
 [No]|Yes: yes
Importing the data ... Done.
```

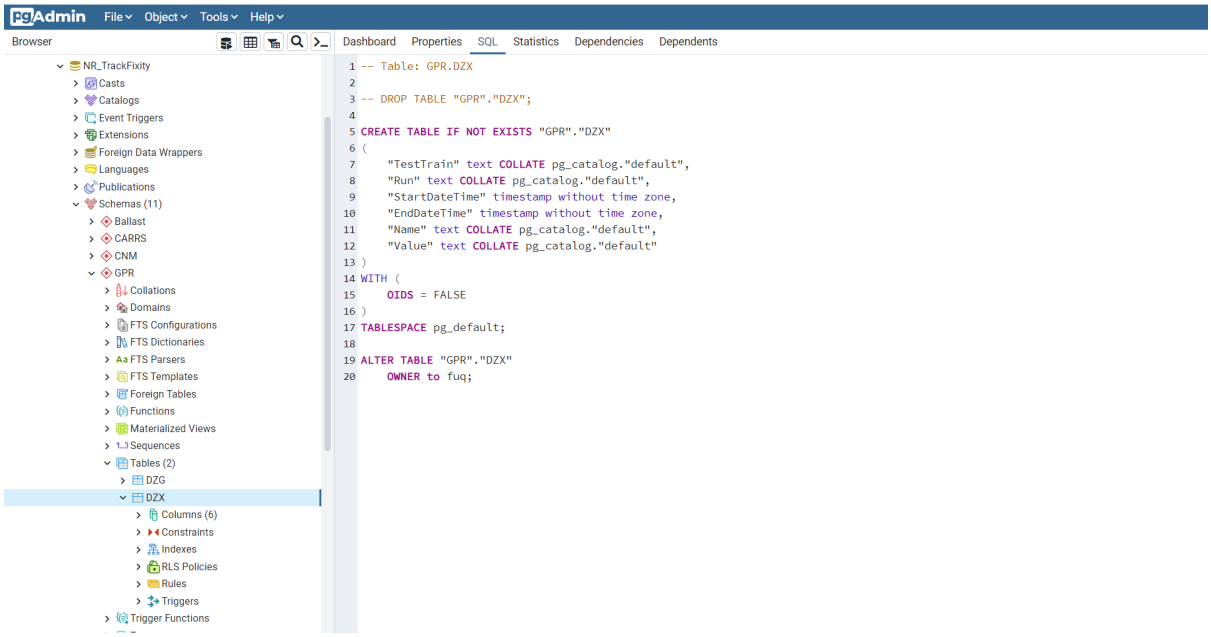


Figure 24: Snapshot of the "GPR"."DZX" table.

GPR.parse_dzg

static GPR.**parse_dzg**(*path_to_dzg*)

TODO: Parse a DZG data file.

Parameters

path_to_dzg (*str*) – Path to a DZG file.

Returns

Parsed DZG data (in tabular form).

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import GPR
>>> from pyhelpers.dirs import cd
>>> gpr = GPR()
>>> # gpr.unzip_backup_data(gpr_date='20200531', verbose=True)
>>> gpr_files_info = gpr.get_files_info(gpr_date='20200531')
>>> k1 = list(gpr_files_info.keys())[0]
>>> dzg_filename = gpr_files_info[k1]['Filenames']['DZG'][0]
>>> dzg_file_path = cd(gpr.FILE_DATA_DIR, "20200531", dzg_filename)
>>> dzg_dat = gpr.parse_dzg(dzg_file_path)
>>> type(dzg_dat)
pandas.core.frame.DataFrame
>>> dzg_dat.shape
(2833, 16)
```

GPR.parse_dzt_

static GPR.**parse_dzt_**(*path_to_dzt*)

TODO: Parse a DZT data file.

Parameters
path_to_dzt (*str*) – Path to a DZT file.

Returns
Parsed DZT data.

Return type
list

Examples:

```
>>> from src.preprocessor import GPR
>>> from pyhelpers.dirs import cd
>>> gpr = GPR()
>>> # gpr.unzip_backup_data(gpr_date='20200531', verbose=True)
>>> gpr_files_info = gpr.get_files_info(gpr_date='20200531')
>>> k1 = list(gpr_files_info.keys())[0]
>>> dzt_filename = gpr_files_info[k1]['Filenames']['DZT'][0]
>>> dzt_file_path = cd(gpr.FILE_DATA_DIR, "20200531", dzt_filename)
>>> dzt_file = gpr.parse_dzt_(path_to_dzt=dzt_file_path)
>>> type(dzt_file)
list
```

GPR.parse_dzx

static GPR.**parse_dzx**(*path_to_dzx*, *head_tag='DZX'*)

Parse a DZX file (.DZX).

Parameters

- **path_to_dzx** (*str*) – Path to a DZX file for GPR data.
- **head_tag** (*str | None*) – Head tag; defaults to `'DZX'`.

Returns
Parsed data of the DZX file.

Return type
pandas.DataFrame

Examples:

```
>>> from src.preprocessor import GPR
>>> from pyhelpers.dirs import cd
>>> gpr = GPR()
>>> # gpr.unzip_backup_data(gpr_date='20200531', verbose=True)
>>> gpr_files_info = gpr.get_files_info(gpr_date='20200531')
>>> k1 = list(gpr_files_info.keys())[0]
>>> dzx_filename = gpr_files_info[k1]['Filenames']['DZX']
>>> dzx_file_path = cd(gpr.FILE_DATA_DIR, "20200531", dzx_filename)
>>> dzx_file = gpr.parse_dzx(path_to_dzx=dzx_file_path)
>>> dzx_file.shape
(12, 2)
```

GPR.parse_gpr_log

static GPR.**parse_gpr_log**(*path_to_gpr_log*, *head_tag='ZRDASV2_GPR_Log'*)

Parse a GPR log file (.lxml).

Parameters

- **path_to_gpr_log** (*str*) – Path to a log file for GPR data.
- **head_tag** (*str | None*) – Name of the top tag; defaults to `'ZRDASV2_GPR_Log'`.

Returns
Parsed data of the GPR log file.

Return type
tuple

Examples:

```
>>> from src.preprocessor import GPR
>>> from pyhelpers.dirs import cd, cdd
>>> import os
>>> gpr = GPR()
>>> # gpr.unzip_data(verbose=True)
>>> gpr_log_file_paths = []
>>> for root, dirs, files in os.walk(cd(gpr.FILE_DATA_DIR, "20200531")):
...     for filename in files:
...         if filename.endswith(".lxml"):
...             gpr_log_file_paths.append(cd(root, filename))
>>> gpr_log_file, chunk_files_info = gpr.parse_gpr_log(gpr_log_file_paths[0])
>>> type(gpr_log_file)
collections.OrderedDict
>>> list(gpr_log_file.keys())
['MetaData', 'System_Parameters', 'Operator_Messages', 'ChunkedFiles']
>>> type(chunk_files_info)
```

(continues on next page)

(continued from previous page)

```
collections.OrderedDict
>>> len(chunk_files_info)
2
```

GPR.read_dzg

GPR.**read_dzg**(*update=False*, *verbose=False*)

Read DZG data.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Returns

DZG data.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import GPR
>>> gpr = GPR()
>>> # gpr.unzip_data(verbose=True)
>>> gpr_dzg_data = gpr.read_dzg()
>>> gpr_dzg_data.shape
(517870, 20)
```

GPR.read_dzg_by_date

GPR.**read_dzg_by_date**(*gpr_date*, *update=False*, *verbose=False*)

Read DZG data for a given date (i.e. data folder).

Parameters

- **gpr_date** (*str*) – Date of the data (i.e. name of the data folder).
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Returns

DZG data for the given date (i.e. data folder).

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import GPR
>>> gpr = GPR()
>>> # gpr.unzip_backup_data(gpr_date='20200531', verbose=True)
>>> dzg_file_data = gpr.read_dzg_by_date(gpr_date='20200531')
>>> dzg_file_data.shape
(73849, 20)
```

GPR.read_dzt_by_date

GPR.**read_dzt_by_date**(*gpr_date*, *update=False*, *pickle_it=False*, *verbose=False*)

Read DZT data for a given date (i.e. data folder).

Parameters

- **gpr_date** (*str*) – Date of the data (i.e. name of the data folder).
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **pickle_it** (*bool*) – Whether to save the DZT data as a pickle file; defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Returns

DZT data for the given date (i.e. data folder).

Return type

list

Examples:

```
>>> from src.preprocessor import GPR
>>> gpr = GPR()
>>> # gpr.unzip_backup_data(gpr_date='20200531', verbose=True)
>>> dzt_file_data = gpr.read_dzt_by_date(gpr_date='20200531')
>>> dzt_file_dat1 = dzt_file_data[0]
>>> type(dzt_file_dat1)
dict
>>> list(dzt_file_dat1.keys())
['DZT', 'TestTrain', 'Run', 'StartDateTime', 'EndDateTime']
>>> type(dzt_file_dat1['DZT'])
list
>>> type(dzt_file_dat1['DZT'][0])
numpy.ndarray
>>> dzt_file_dat1['DZT'][0].shape
(512, 9823)
>>> type(dzt_file_dat1['DZT'][1])
numpy.ndarray
>>> dzt_file_dat1['DZT'][1].shape
(512, 9823)
```

GPR.read_dzx

GPR.**read_dzx**(*update=False*, *verbose=False*)

Read DZX data.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Returns

DZX data.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import GPR
>>> gpr = GPR()
>>> # gpr.unzip_data(verbose=True)
>>> gpr_dzx_data = gpr.read_dzx()
>>> gpr_dzx_data.shape
(180, 6)
```

GPR.read_dzx_by_date

GPR.**read_dzx_by_date**(*gpr_date*, *update=False*, *verbose=False*)

Read DZX data for a given date (i.e. data folder).

Parameters

- **gpr_date** (*str*) – Date of the data (i.e. name of the data folder).
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Returns

DZX data for the given date (i.e. data folder).

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import GPR
>>> gpr = GPR()
>>> # gpr.unzip_backup_data(gpr_date='20200531', verbose=True)
>>> dzx_file_data = gpr.read_dzx_by_date(gpr_date='20200531')
```

(continues on next page)

(continued from previous page)

```
>>> dzx_file_data.shape
(60, 6)
```

GPR.unzip_backup_data

GPR.**unzip_backup_data**(*gpr_date*, *confirmation_required=True*, *verbose=False*)

Unzip GPR data files (up to the given date) in a local directory.

Parameters

- **gpr_date** (*str*) – Date of the data (i.e., name of the data folder).
- **confirmation_required** (*bool*) – Whether confirmation is required to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Examples:

```
>>> from src.preprocessor import GPR
>>> gpr = GPR()
>>> gpr.unzip_backup_data(gpr_date='20200531', verbose=True)
To extract the GPR data of "20200531" from its zipped files
? [No]|Yes: yes
Extraction in progress ...
    Phase 1: extracting the original zipped data files ... Done.
    Phase 2: extracting further the decompressed files ... Done.
```

GPR.unzip_data

GPR.**unzip_data**(*confirmation_required=True*, *verbose=False*)

Unzip all available GPR data files from the backup in a local directory.

Parameters

- **confirmation_required** (*bool*) – Whether confirmation is required to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Examples:

```
>>> from src.preprocessor import GPR
>>> gpr = GPR()
>>> gpr.unzip_data(verbose=True)
To extract all available GPR data from their zipped files
? [No]|Yes: yes
Extraction in progress ...
    "20200107" ... Done.
    "20200110" ... Done.
```

(continues on next page)

(continued from previous page)

```
    "20200116" ... Done.
    "20200312" ... Done.
    "20200428" ... Done.
    "20200531" ... Done.
```

GPR.visualise_dzt_

GPR.**visualise_dzt_**(*path_to_dzt*, *channel=0*)

TODO: Visualise DZT data.

Parameters

- **path_to_dzt** (*str*) – Path to a DZT file.
- **channel** (*int*) – Channel number to visualise; defaults to 0.

2.2.6 INM

class src.preprocessor.**INM**(*db_instance=None*)

Integrated Network Model.

Note: This class currently handles only the INM combined data report, which is available as a CSV file.

Parameters

db_instance (TrackFixityDB | *None*) – PostgreSQL database instance; defaults to None.

Variables

- **cdr_dtypes_object** (*dict*) – Field types for object data in the INM CDR.
- **cdr_dtypes_float** (*dict*) – Field types for float data in the INM CDR.
- **cdr_dtypes_int** (*dict*) – Field types for integer data in the INM CDR.
- **cdr_dtypes_mix** (*dict*) – Field types for mixed-type data in the INM CDR, potentially containing errors.
- **cdr_dtypes** (*dict*) – Field types of the INM CDR data.
- **mileage_dtypes** (*dict*) – Field types for mileage data in the INM CDR.
- **db_instance** (TrackFixityDB) – PostgreSQL database instance for database operations.

Examples:

```
>>> from src.preprocessor import INM
>>> inm = INM()
>>> inm.NAME
'Integrated Network Model'
```

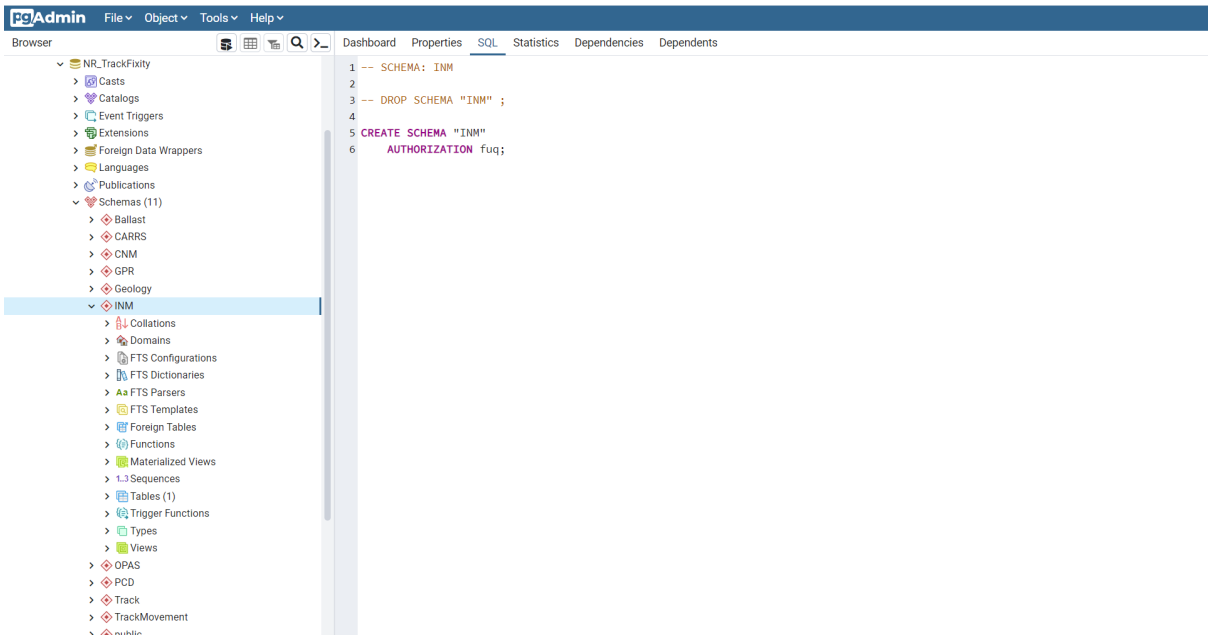


Figure 25: Snapshot of the *INM* schema.

Attributes:

ACRONYM	Acronym for the data name.
CDR_FILENAME	Filename of the original INM CDR data.
CDR_NAME	Name of the INM combined data report (CDR).
CDR_TABLE_NAME	Name of table for storing the data of INM combined data report.
DATA_DIR	Pathname of a local directory where the INM data is stored.
NAME	Data name.
SCHEMA_NAME	Name of schema for storing the INM data.

INM.ACRONYM

INM.**ACRONYM: str = 'INM'**

Acronym for the data name.

INM.CDR_FILENAME

INM.**CDR_FILENAME: str = 'INM combined data report'**

Filename of the original INM CDR data.

INM.CDR_NAME

INM.**CDR_NAME: str = 'Combined data report'**

Name of the INM combined data report (CDR).

INM.CDR_TABLE_NAME

INM.**CDR_TABLE_NAME: str = 'Combined data report'**

Name of table for storing the data of INM combined data report.

INM.DATA_DIR

INM.**DATA_DIR: str = 'data\\INM'**

Pathname of a local directory where the INM data is stored.

INM.NAME

INM.**NAME: str = 'Integrated Network Model'**

Data name.

INM.SCHEMA_NAME

INM.**SCHEMA_NAME: str = 'INM'**

Name of schema for storing the INM data.

Methods:

import_combined_data_report([update, ...])	Import the data from the Integrated Network Model (INM) combined data report into the project database.
load_combined_data_report([elr])	Load the data of the Integrated Network Model (INM) combined data report from the project database.
read_combined_data_report([update, verbose])	Read INM combined data report from a local directory.

INM.import_combined_data_report

INM.**import_combined_data_report**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import the data from the Integrated Network Model (INM) combined data report into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **confirmation_required** (*bool*) – Whether to ask for confirmation before proceeding; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters to pass to the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import INM
>>> inm = INM()
>>> inm.import_combined_data_report()
To import INM combined data report into the table "INM"."Combined data report"?
 [No]|Yes: yes
Importing the data ... Done.
```

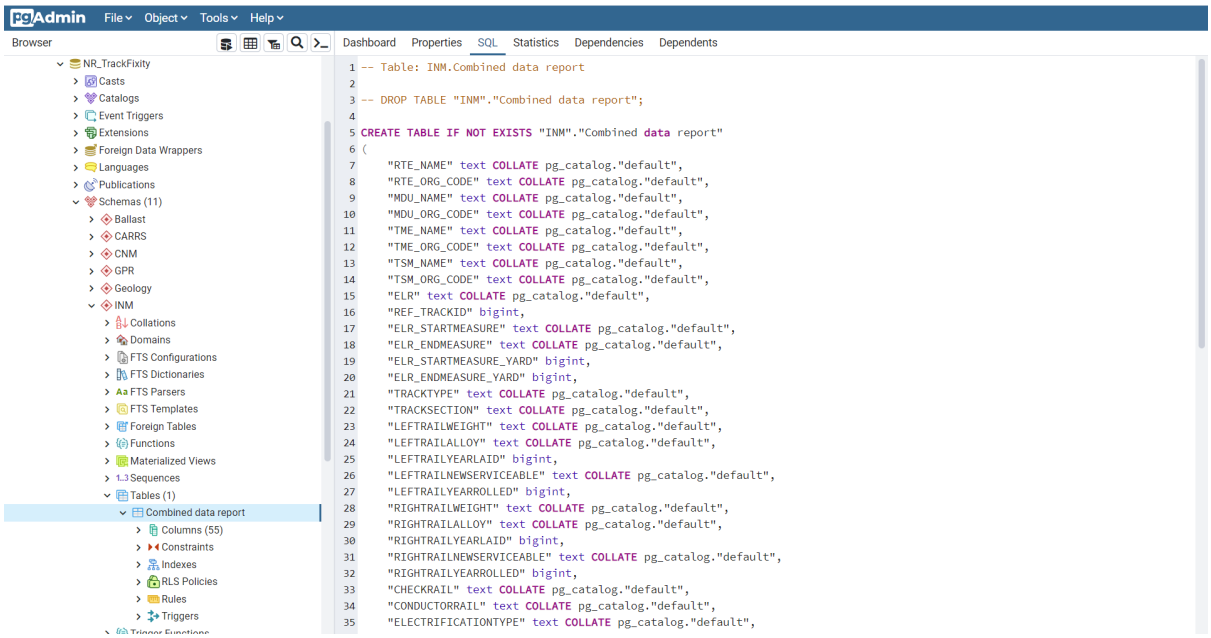


Figure 26: Snapshot of the "INM"."Combined data report" table.

INM.load_combined_data_report

INM.**load_combined_data_report**(*elr=None*, ***kwargs*)

Load the data of the Integrated Network Model (INM) combined data report from the project database.

Parameters

- **elr** (*str | list | tuple | None*) – Engineer's Line Reference(s) to filter the data; defaults to None.
- **kwargs** – [Optional] additional parameters to pass to the method pyhelpers.dbms.PostgreSQL.read_sql_query.

Returns

Data of the INM combined data report.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import INM
>>> inm = INM()
>>> inm_cdr = inm.load_combined_data_report(elr=['ECM7', 'ECM8'])
>>> inm_cdr.shape
(5510, 55)
```

INM.read_combined_data_report

INM.**read_combined_data_report**(*update=False*, *verbose=False*)

Read INM combined data report from a local directory.

Parameters

- **update** (*bool*) – Whether to re-read the original INM combined data report file; defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Returns

Data of the INM combined data report.

Return type

pandas.DataFrame | None

Examples:

```
>>> from src.preprocessor import INM
>>> inm = INM()
>>> inm_cdr = inm.read_combined_data_report()
>>> inm_cdr.shape
(486674, 55)
```

Note:

- `'SLEEPERSPER60FOOTLENGTH'` (`int`): strings (i.e. `'PAN'` and `'NONE'`) and `nan`
- `'PATCHPROGRAM'` (`str` of 2 digits of int): 2019, `'Patch Program'` and `nan`
- `'TRACK_CATEGORY'` (`int`, 1-6): `nan` and `'1A'`
- `'SMOOTH_TRACK_CAT'` (`int`, 1-6): `nan` and `'1A'`

2.2.7 OPAS

class src.preprocessor.**OPAS**(*db_instance=None*)

Operational Property Asset System.

Note: This class currently handles only the DGN data about stations.

Parameters

db_instance (TrackFixityDB | *None*) – PostgreSQL database instance; defaults to `None`.

Variables

db_instance (TrackFixityDB) – PostgreSQL database instance for database operations.

Examples:

```
>>> from src.preprocessor import OPAS
>>> opas = OPAS()
>>> opas.NAME
'Operational Property Asset System'
```

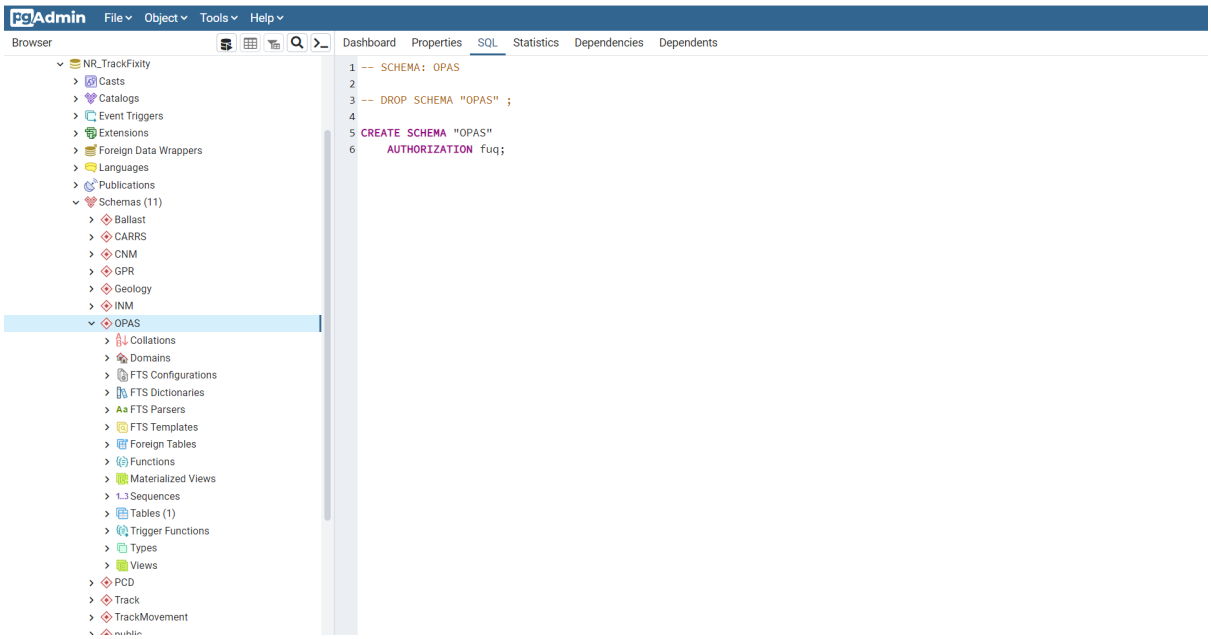


Figure 27: Snapshot of the *OPAS* schema.

Attributes:

ACRONYM	Acronym for the data name.
DATA_DIR	Pathname of a local directory where the OPAS data is stored.
NAME	Data name.
PROJ_DIRNAME	Directory name of projection/boundary shapefiles.
PROJ_FILENAME	Name of the projection file.
SCHEMA_NAME	Name of the schema for storing the OPAS data.
STN_DIRNAME	Directory name of *stations* data.
STN_TABLE_NAME	Name of the table for storing the *stations* data.

OPAS.ACRONYM

OPAS.**ACRONYM: str = 'OPAS'**

Acronym for the data name.

OPAS.DATA_DIR

OPAS.**DATA_DIR: str = 'data\\OPAS'**

Pathname of a local directory where the OPAS data is stored.

OPAS.NAME

OPAS.**NAME: str = 'Operational Property Asset System'**

Data name.

OPAS.PROJ_DIRNAME

OPAS.**PROJ_DIRNAME: str = 'Projection'**

Directory name of projection/boundary shapefiles.

OPAS.PROJ_FILENAME

OPAS.**PROJ_FILENAME: str = 'Order_69356_Polygon'**

Name of the projection file.

OPAS.SCHEMA_NAME

OPAS.**SCHEMA_NAME: str = 'OPAS'**

Name of the schema for storing the OPAS data.

OPAS.STN_DIRNAME

OPAS.**STN_DIRNAME: str = 'Stations'**

Directory name of *stations* data.

OPAS.STN_TABLE_NAME

OPAS.**STN_TABLE_NAME: str = 'Stations'**

Name of the table for storing the *stations* data.

Methods:

dgn2shp([dat_name, confirmation_required, ...])	Convert DGN files of OPAS data to shapefiles.
import_stations_shp([update, ...])	Import shapefile data (converted from the DGN data) of stations into the project database.
load_stations_shp(**kwargs)	Load the shapefile data (converted from the DGN data) of stations from the project database.
map_view([item_name, layer_name, ...])	Make a map view of a given item.
read_prj_metadata([update, parser, as_dict, ...])	Read the projection metadata from DGN files or shapefiles.
read_stations_shp([update, verbose])	Read the shapefile of stations (converted from DGN data) from a local directory.

OPAS.dgn2shp

OPAS.**dgn2shp**(*dat_name='Stations'*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Convert DGN files of OPAS data to shapefiles.

Parameters

- **dat_name** (*str | None*) – Name of the data; defaults to `'Stations'`.
- **confirmation_required** (*bool*) – Whether to ask for confirmation to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the `dgn2shp()` function.

Examples:

```
>>> from src.preprocessor import CNM
>>> opas = OPAS()
>>> opas.dgn2shp()
To convert .dgn files of "Stations" to shapefiles?
 [No]|Yes: yes
Converting "ENG_ADMIN.RINM_OPAS_Stations.dgn" at "data\OPAS\Stations" ... Done.
```

OPAS.import_stations_shp

OPAS.**import_stations_shp**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import shapefile data (converted from the DGN data) of stations into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **confirmation_required** (*bool*) – Whether to ask for confirmation to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import OPAS
>>> opas = OPAS()
>>> opas.import_stations_shp(if_exists='replace')
To import shapefile of stations into the table "OPAS"."Stations"?
 [No]|Yes: yes
Importing the data ... Done.
```

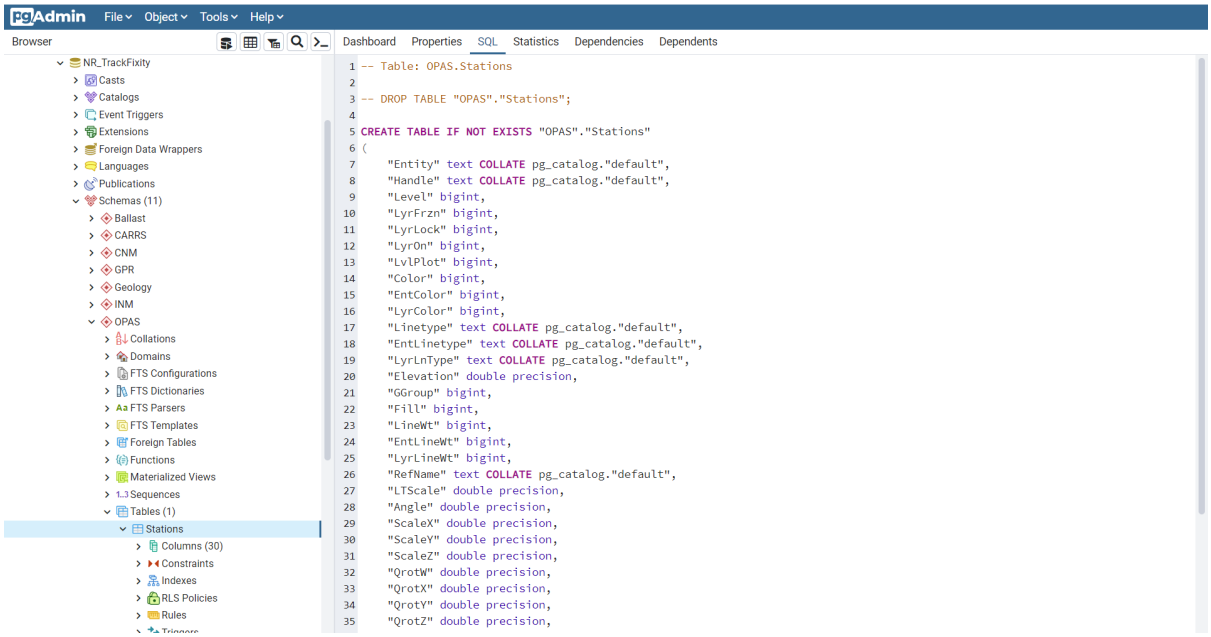


Figure 28: Snapshot of the "OPAS"."Stations" table.

OPAS.load_stations_shp

OPAS.**load_stations_shp**(***kwargs*)

Load the shapefile data (converted from the DGN data) of stations from the project database.

Parameters

kwargs – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.read_sql_query.

Returns

DGN-converted shapefile data of stations.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import OPAS
>>> opas = OPAS()
>>> stn_shp = opas.load_stations_shp()
>>> stn_shp.head()
     Entity  ...                                          geometry
0     Point  ...  POINT Z (164858.9644999998 39701.62969999947 0)
1     Point  ...  POINT Z (167561.3097000001 797058.6073000003 0)
2     Point  ...        POINT Z (176248.3465 827083.7353000008 0)
3     Point  ...  POINT Z (181803.4960000003 32372.75510000065 0)
4     Point  ...  POINT Z (190012.4675000003 206251.4306000005 0)
[5 rows x 30 columns]
```

OPAS.map_view

OPAS.**map_view**(*item_name='Stations'*, *layer_name='Point'*, *desc_col_name='Handle'*, *sample=True*, *marker_colour='darkred'*, *update=False*, *verbose=True*)

Make a map view of a given item.

Parameters

- **item_name** (*str*) – Name of the item; defaults to `'Stations'`.
- **layer_name** (*str*) – Name of the layer; defaults to `'Point'`.
- **desc_col_name** (*str*) – Name of the column that describes markers; defaults to `'Handle'`.
- **sample** (*bool | int*) – Whether to draw a sample or specify a sample size.
- **marker_colour** (*str*) – Colour of the markers; defaults to `'darkred'`.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.

- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `True`.

Examples:

```
>>> from src.preprocessor import OPAS
>>> opas = OPAS()
>>> opas.map_view(desc_col_name='Handle', sample=100)
```

Figure 29: Eamples of stations.

OPAS.read_prj_metadata

OPAS.**read_prj_metadata**(*update=False*, *parser='osr'*, *as_dict=True*, *verbose=False*)

Read the projection metadata from DGN files or shapefiles.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **parser** (*str*) – Package used for reading the PRJ file; options are `'osr'` or `'pycrs'`; defaults to `'osr'`.
- **as_dict** (*bool*) – Whether to return the data as a dictionary; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `False`.

Returns

Projection metadata from the DGN files or shapefiles.

Return type

dict | None

Examples:

```
>>> from src.preprocessor import OPAS
>>> opas = OPAS()
>>> dgn_shp_prj = opas.read_prj_metadata()
>>> type(dgn_shp_prj)
dict
>>> list(dgn_shp_prj.keys())
['PROJCS', 'Shapefile']
>>> list(dgn_shp_prj['PROJCS'].keys())
['proj', 'lat_0', 'lon_0', 'k', 'x_0', 'y_0', 'ellps', 'units']
>>> dgn_shp_prj['Shapefile']
  ORDER_ID  ...                                           geometry
0    69356  ...  POLYGON ((380323.430 961539.720, 698840.025 24...
[1 rows x 4 columns]
```

OPAS.read_stations_shp

OPAS.**read_stations_shp**(*update=False*, *verbose=False*)

Read the shapefile of stations (converted from DGN data) from a local directory.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `False`.

Returns

Shapefile data of stations converted from DGN files.

Return type
dict

Examples:

```
>>> from src.preprocessor import OPAS
>>> opas = OPAS()
>>> stn_shp = opas.read_stations_shp()
>>> type(stn_shp)
dict
>>> list(stn_shp.keys())
['Annotation', 'MultiPatch', 'Point', 'Polygon', 'Polyline']
>>> stn_shp['Annotation'].empty
True
>>> stn_shp['MultiPatch'].empty
True
>>> stn_shp['Point'].empty
False
>>> stn_shp['Polygon'].empty
True
>>> stn_shp['Polyline'].empty
True
>>> stn_shp['Point'].head()
     Entity  ...                                geometry
0     Point  ...   POINT Z (164858.964 39701.630 0.000)
1     Point  ...  POINT Z (167561.310 797058.607 0.000)
2     Point  ...  POINT Z (176248.346 827083.735 0.000)
3     Point  ...   POINT Z (181803.496 32372.755 0.000)
4     Point  ...  POINT Z (190012.468 206251.431 0.000)
[5 rows x 30 columns]
```

2.2.8 PCD

class src.preprocessor.**PCD**(*elr='ECM8'*, *db_instance=None*)

Point cloud data.

In this project, there are three types of point cloud data (PCD), including:

- raw data (.LAZ/.LAS) from scanners
- data in DGN format
- data in KRDZ format

Parameters

- **elr** (*str*) – Engineer's Line Reference; defaults to `'ECM8'`.
- **db_instance** (TrackFixityDB | *None*) – PostgreSQL database instance; defaults to `None`.

Variables

- **elr** (*str*) – Engineer's Line Reference.
- **dgn_dir** (*Callable*) – Pathname of the local directory for storing DGN files (for a given date).

- **laz_dir** (*Callable*) – Pathname of the local directory for storing LAZ files (for a given date).
- **krdz_dir** (*Callable*) – Pathname of the local directory for storing KRDZ files (for a given date).
- **data_dates** (*list*) – Dates of the available data.
- **dgn_filename** (*Callable*) – Filename of DGN data.
- **dgn_shp_filename** (*Callable*) – Filename of shapefile data converted from a DGN file.
- **krdz_filename** (*Callable*) – Filename of KRDZ data.
- **db_instance** (*pyhelpers.dbms.PostgreSQL*) – PostgreSQL database instance.
- **laz_table_name** (*Callable*) – Name of the table storing the LAZ data for a given date.
- **laz_meta_table_name** (*Callable*) – Name of the table storing the metadata of the LAZ data for a given date.
- **dgn_shp_table_name** (*Callable*) – Name of the table storing the DGN-converted shapefiles.

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> pcd.NAME
'Point cloud data'
>>> pcd.SCHEMA_NAME
'PCD'
```

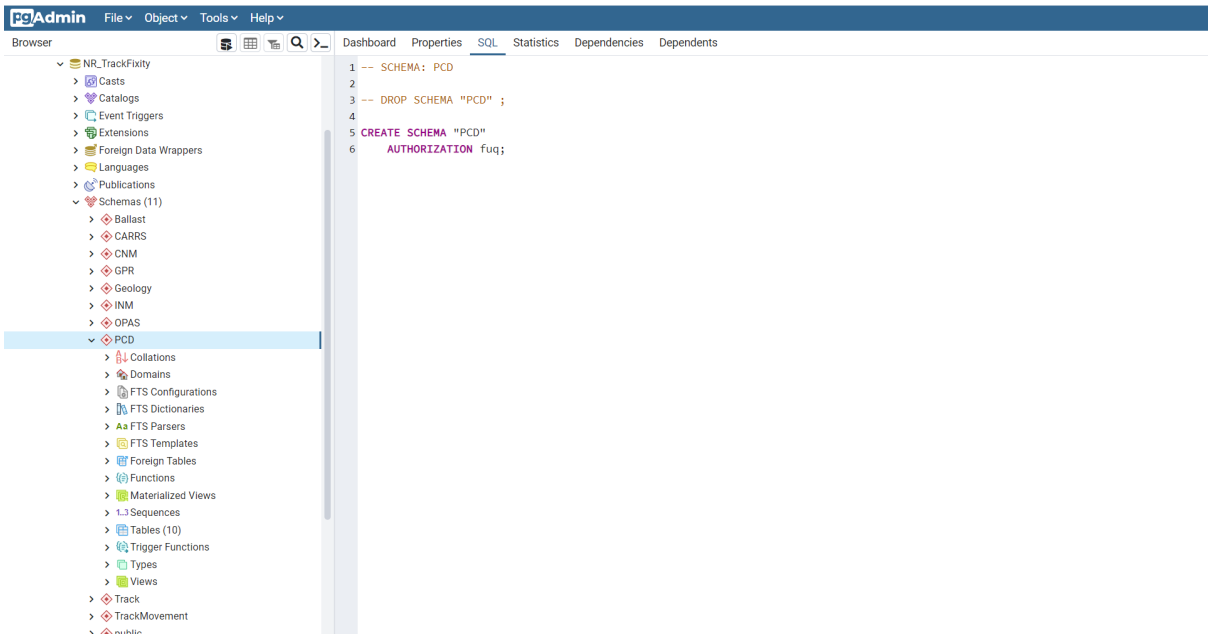


Figure 30: Snapshot of the *PCD* schema.

Attributes:

ACRONYM	Acronym for the data name.
DATA_DIR	Pathname of a local directory where the point cloud data is stored.
KRDZ_CL_TABLE_NAME	Name of the table storing the classified KRDZ data.
KRDZ_META_TABLE_NAME	Name of the table storing the metadata of KRDZ data.
KRDZ_SCHEMA_FILENAME	Filename of the KRDZ schema file.
KRDZ_SCHEMA_NAME	Name of the schema for storing the KRDZ data.
KRDZ_TABLE_NAME	Name of the table storing the KRDZ data.
LAZ_META_TABLE_NAME	Name of the table storing the metadata of the LAZ data.
LAZ_TABLE_NAME	Name of the table storing the LAZ data.
MILEAGE	Short description of start and end mileages.
NAME	Data name.
SCHEMA_NAME	Name of the schema for storing the point cloud data.
TILES_FILENAME	Filename of tiles data.
TILES_META_TABLE_NAME	Name of the table storing the metadata of the tile data.
TILES_TABLE_NAME	Name of the table storing the tile data.

PCD.ACRONYM

PCD.**ACRONYM: str = 'PCD'**

Acronym for the data name.

PCD.DATA_DIR

PCD.**DATA_DIR: str = 'data\\PCD'**

Pathname of a local directory where the point cloud data is stored.

PCD.KRDZ_CL_TABLE_NAME

PCD.**KRDZ_CL_TABLE_NAME: str = 'KRDZ_Classified'**

Name of the table storing the classified KRDZ data.

PCD.KRDZ_META_TABLE_NAME

PCD.**KRDZ_META_TABLE_NAME: str = 'KRDZ_Metadata'**

Name of the table storing the metadata of KRDZ data.

PCD.KRDZ_SCHEMA_FILENAME

PCD.**KRDZ_SCHEMA_FILENAME: str = 'KRDZ_schema'**

Filename of the KRDZ schema file.

PCD.KRDZ_SCHEMA_NAME

PCD.**KRDZ_SCHEMA_NAME: str = 'PCD'**

Name of the schema for storing the KRDZ data.

PCD.KRDZ_TABLE_NAME

PCD.**KRDZ_TABLE_NAME: str = 'KRDZ'**

Name of the table storing the KRDZ data.

PCD.LAZ_META_TABLE_NAME

PCD.**LAZ_META_TABLE_NAME: str = 'LAZ_OSGB_100x100_Metadata'**

Name of the table storing the metadata of the LAZ data.

PCD.LAZ_TABLE_NAME

PCD.**LAZ_TABLE_NAME: str = 'LAZ_OSGB_100x100'**

Name of the table storing the LAZ data.

PCD.MILEAGE

PCD.**MILEAGE: str = 'ECM7_67m_69m67ch_ECM8_10m_54m50ch'**

Short description of start and end mileages.

PCD.NAME

PCD.**NAME: str = 'Point cloud data'**

Data name.

PCD.SCHEMA_NAME

PCD.**SCHEMA_NAME: str = 'PCD'**

Name of the schema for storing the point cloud data.

PCD.TILES_FILENAME

PCD.**TILES_FILENAME: str = 'project.prj'**

Filename of tiles data.

PCD.TILES_META_TABLE_NAME

PCD.**TILES_META_TABLE_NAME: str = 'Tiles_Metadata'**

Name of the table storing the metadata of the tile data.

PCD.TILES_TABLE_NAME

PCD.**TILES_TABLE_NAME: str = 'Tiles'**

Name of the table storing the tile data.

Methods:

check_pcd_dates(pcd_dates[, len_req])	Check the dates provided as input for point cloud data.
dgn2shp([pcd_date, confirmation_required, ...])	Convert DGN data to shapefiles for a specific date of the point cloud data.
flatten_dgn_pcd(dgn_pl_dat)	Flatten the polyline geometry data of the DGN-converted shapefile.
import_dgn_shp([confirmation_required, ...])	Import shapefile data (converted from DGN data) into the project database.
import_krdz([update, confirmation_required, ...])	Import KRDZ data into the project database.
import_krdz_metadata([update, ...])	Import metadata for the KRDZ data into the project database.
import_laz([subset, incl_metadata, ...])	Import LAZ data into the project database.
import_laz_by_date(pcd_date[, subset, ...])	Import LAZ data for a given date into the project database.
import_tiles([update, ...])	Import information about tiles into the project database.
load_dgn_shp([pcd_date, layer_name])	Load data of the DGN-converted shapefile from the project database.
load_krdz([pcd_dates])	Load KRDZ data (for a given data date or dates) from the project database.
load_laz(tile_xy[, pcd_dates, greyscale, ...])	Load LAZ data within a given tile and dates of the point cloud data from the project database.
load_tiles([pcd_date])	Load data of the tiles for point cloud data.
map_view_tiles([tile_colours, update, verbose])	Make a map view of tiles of all available point cloud data.
map_view_tiles_by_date(pcd_date[, ...])	Make a map view of tiles of point cloud data for a given date.
parse_laz(path_to_laz[, ...])	Transform LAS/LAZ data into a dataframe.
read_dgn_shp([update, verbose])	Read all available DGN-converted shapefiles from a local directory.
read_dgn_shp_by_date([pcd_date, update, ...])	Read DGN-converted shapefiles for a specific date of the point cloud data.
read_krdz([update, verbose])	Read a KRDZ file from a local directory.
read_krdz_by_date(pcd_date[, update, verbose])	Read a KRDZ file for a given date from a local directory.
read_krdz_metadata([update, verbose])	Read the metadata for the KRDZ data from a local directory.
read_tiles_prj([update, verbose])	Read all PRJ files for tiles from a local directory.
read_tiles_prj_by_date(pcd_date[, update, ...])	Read a PRJ file of tiles for a given date from a local directory.

PCD.check_pcd_dates

PCD.**check_pcd_dates**(*pcd_dates*, *len_req=None*)

Check the dates provided as input for point cloud data.

Parameters

- **pcd_dates** (*int | str | list | tuple | None*) – Dates of the point cloud data.
- **len_req** (*int | None*) – Length required for the range of the pcd_dates; defaults to None.

Returns

List of validated data dates.

Return type

list

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> pcd.check_pcd_dates('201910')
['201910']
>>> pcd.check_pcd_dates(['201910', 202004])
['201910', '202004']
```

PCD.dgn2shp

PCD.**dgn2shp**(*pcd_date=None*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Convert DGN data to shapefiles for a specific date of the point cloud data.

Parameters

- **pcd_date** (*str | None*) – Date of the point cloud data, e.g., '201910', '202004'. When pcd_date=None (default), the function converts all available "ECM8_10_66_*_20200625" DGN data.
- **confirmation_required** (*bool*) – Whether to ask for confirmation to proceed; defaults to True.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to True.
- **kwargs** – [Optional] additional parameters for the function dgn2shp().

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> pcd.dgn2shp()
To convert the DGN file "ECM8_10_66_201910_20200625.dgn" to shapefiles?
```

(continues on next page)

(continued from previous page)

```
 [No]|Yes: yes
Converting "ECM8_10_66_201910_20200625.dgn" at "data\PCD\ECM8\DGN\201910" ...␣
 ↪Done.
To convert the DGN file "ECM8_10_66_202004_20200625.dgn" to shapefiles?
 [No]|Yes: yes
Converting "ECM8_10_66_202004_20200625.dgn" at "data\PCD\ECM8\DGN\202004" ...␣
 ↪Done.
```

PCD.flatten_dgn_pcd

static PCD.**flatten_dgn_pcd**(*dgn_pl_dat*)

Flatten the polyline geometry data of the DGN-converted shapefile.

Parameters

dgn_pl_dat (*pandas.DataFrame*) – Polyline geometry data of the DGN-converted shapefile.

Returns

Flattened point coordinates.

Return type

numpy.ndarray

Examples:

```
>>> from src.preprocessor import PCD
>>> from src.utils.general import TrackFixityDB
>>> from pyhelpers.settings import np_preferences
>>> project_db = TrackFixityDB()
>>> pcd = PCD(db_instance=project_db)
>>> tbl_name = pcd.dgn_shp_table_name_('Polyline')
>>> query = f'SELECT * FROM "PCD"."{tbl_name}" WHERE "Year"=2020 AND "Month"=4␣
 ↪LIMIT 5'
>>> example_dat = pcd.db_instance.read_sql_query(query)
>>> example_dat.head()
   Year  Month  ... QrotZ                                           geometry
0  2020      4  ...     0  LINESTRING Z (340183.475 674108.682 33.237, 34...
1  2020      4  ...     0  LINESTRING Z (340184.412 674109.031 33.235, 34...
2  2020      4  ...     0  LINESTRING Z (340185.349 674109.38 33.232, 340...
3  2020      4  ...     0  LINESTRING Z (340233.146 674127.169 33.117, 34...
4  2020      4  ...     0  LINESTRING Z (340234.083 674127.518 33.114, 34...
[5 rows x 28 columns]
>>> np_preferences()
>>> pcd.flatten_dgn_pcd(example_dat)
array([[340183.4750, 674108.6820, 33.2370],
       [340183.4750, 674108.6820, 33.2550],
       [340184.4120, 674109.0310, 33.2350],
       [340184.4120, 674109.0310, 33.2530],
       [340185.3490, 674109.3800, 33.2320],
       [340185.3490, 674109.3800, 33.2500],
       [340233.1460, 674127.1690, 33.1170],
       [340233.1460, 674127.1690, 33.1350],
       [340234.0830, 674127.5180, 33.1140],
       [340234.0830, 674127.5180, 33.1320]])
```

PCD.import_dgn_shp

PCD.**import_dgn_shp**(*confirmation_required=True*, *verbose=True*, *rm_dgn_shp_files=False*, ***kwargs*)

Import shapefile data (converted from DGN data) into the project database.

Parameters

- **confirmation_required** (*bool*) – Whether to ask for confirmation to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `True`.
- **rm_dgn_shp_files** (*bool*) – Whether to remove the converted shapefiles after importing; defaults to `False`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> pcd.import_dgn_shp(if_exists='replace')
To import DGN-shapefiles into the schema "PCD"?
 [No]|Yes: yes
Importing the data:
    Annotation ... Done.
    Polyline ... Done.
```

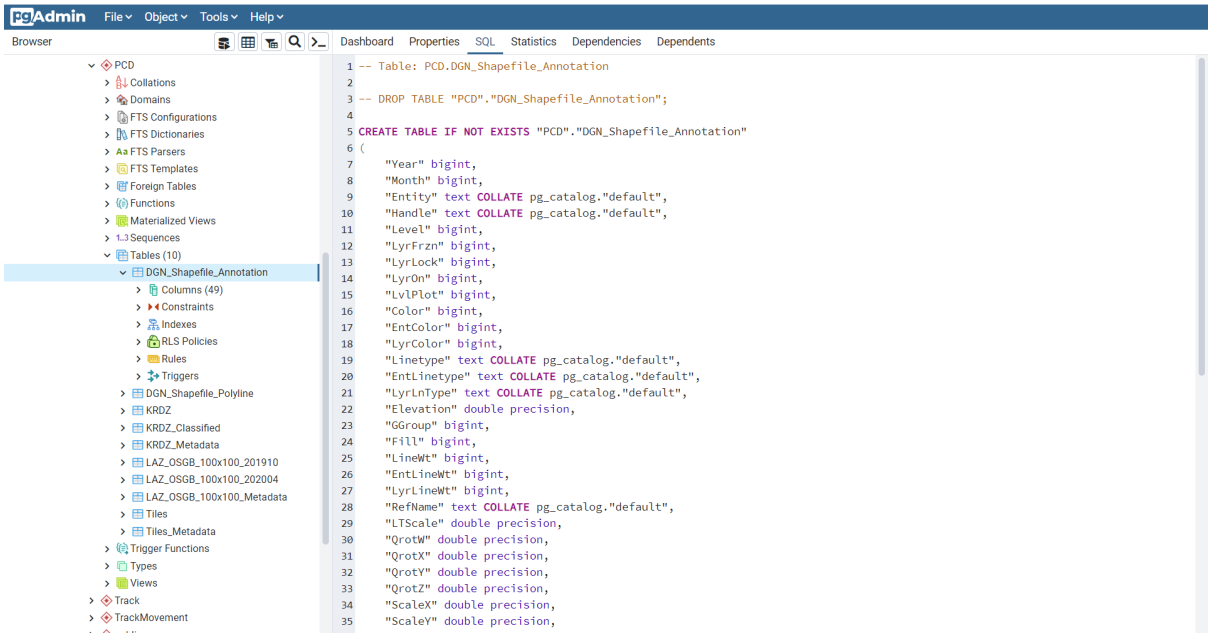


Figure 31: Snapshot of the "PCD"."DGN_Shapefile_Annotation" table.

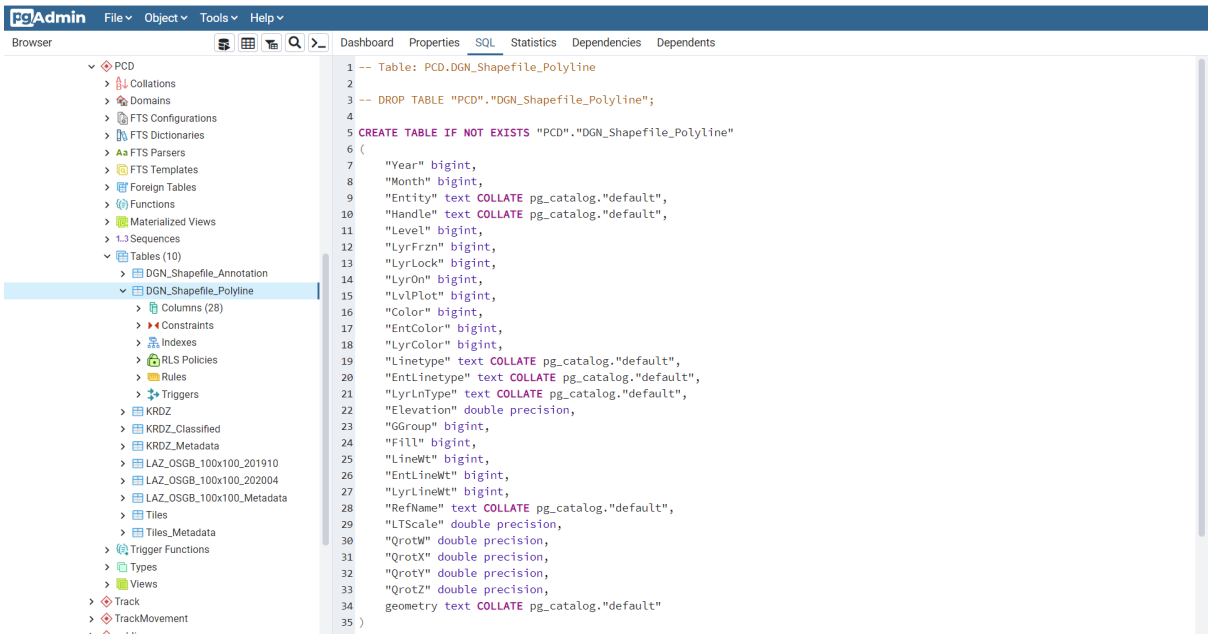


Figure 32: Snapshot of the "PCD"."DGN_Shapefile_Polyline" table.

PCD.import_krdz

PCD.**import_krdz**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import KRDZ data into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **confirmation_required** (*bool*) – Whether to ask for confirmation to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the method *pyhelpers.dbms.PostgreSQL.import_data*.

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> pcd.import_krdz(if_exists='replace')
To import KRDZ data of ECM8 into the table "PCD"."KRDZ"?
 [No]|Yes: yes
Importing the data ... Done.
```

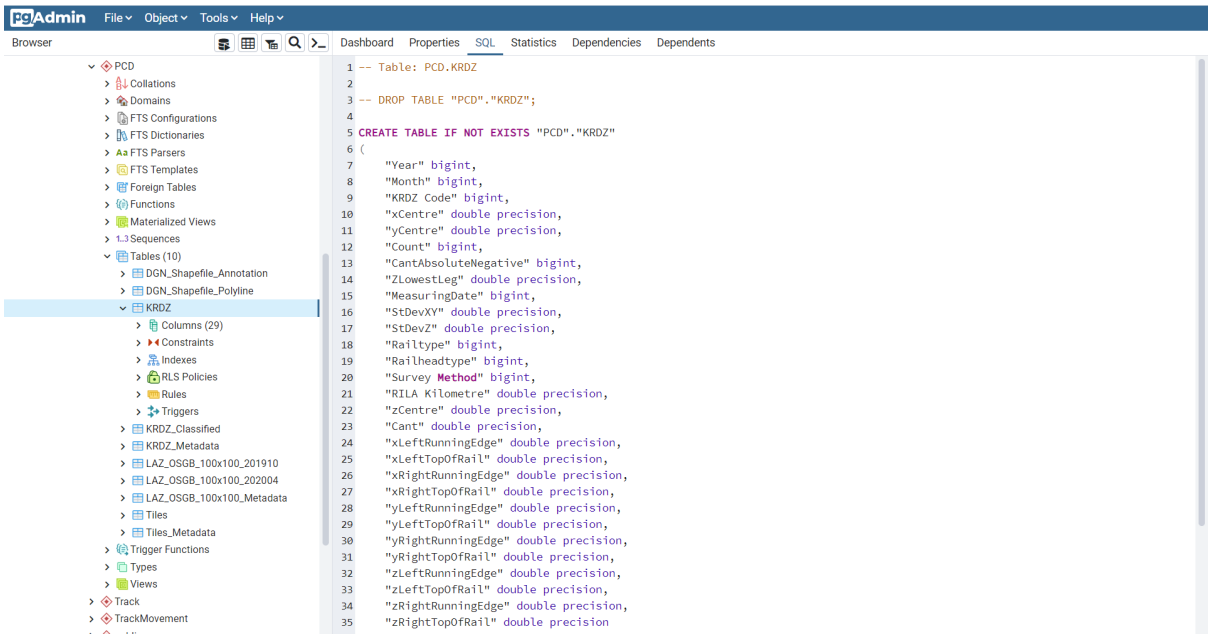


Figure 33: Snapshot of the "PCD"."KRDZ" table.

PCD.import_krdz_metadata

PCD.**import_krdz_metadata**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import metadata for the KRDZ data into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **confirmation_required** (*bool*) – Whether to ask for confirmation to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> pcd.import_krdz_metadata()
To import KRDZ metadata into the table "PCD"."KRDZ_Metadata"?
 [No]|Yes: yes
Importing the data ... Done.
```

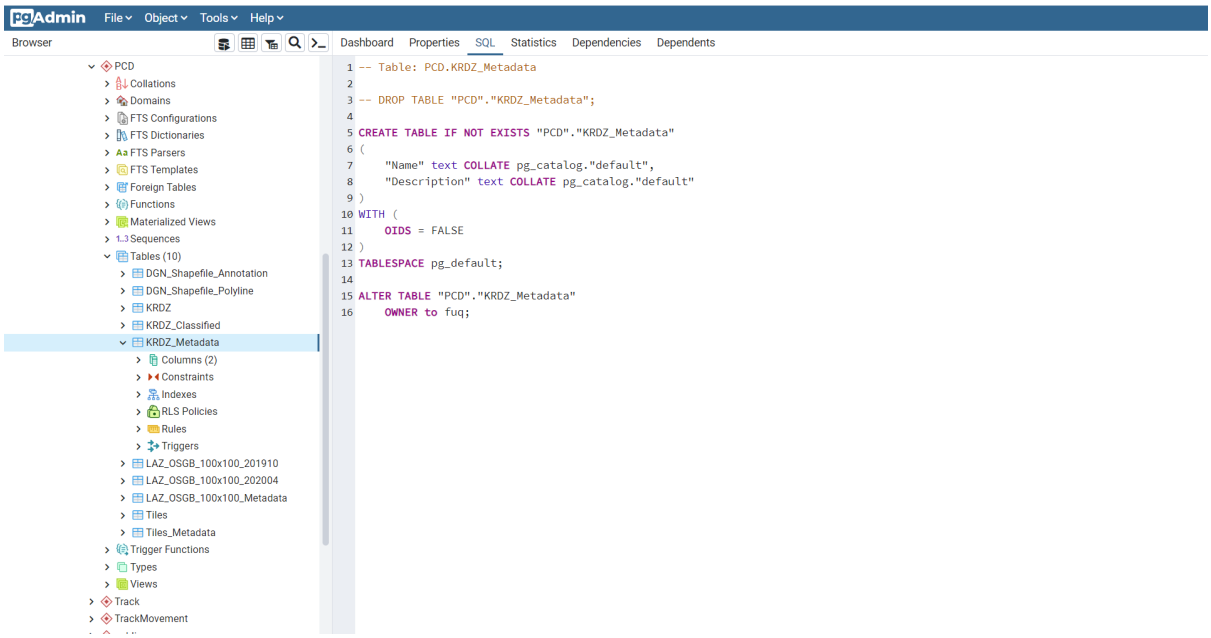


Figure 34: Snapshot of the "PCD"."KRDZ_Metadata" table.

PCD.import_laz

PCD.**import_laz**(*subset=None*, *incl_metadata=True*, *only_metadata=False*, *if_exists='append'*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import LAZ data into the project database.

Parameters

- **subset** (*list | tuple | numpy.ndarray | range | None*) – List of indices indicating specific local file paths to import; defaults to `None`.
- **incl_metadata** (*bool*) – Whether to import the metadata along with the data; defaults to `True`.
- **only_metadata** (*bool*) – Whether to import only the metadata and not the data itself; defaults to `False`.
- **if_exists** (*str*) – Action to take if the targeted table already exists (see pandas.DataFrame.to_sql); defaults to `'append'`.
- **confirmation_required** (*bool*) – Whether to ask for confirmation before proceeding; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Test (201910):

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> pcd.import_laz(if_exists='append')
To import LAZ data into the schema "PCD"?
 [No]|Yes: yes
Importing the data into "PCD"."LAZ_OSGB_100x100" ...
    1/3981: "Tile_X+0000340100_Y+0000674000.laz" ... Done.
    ...
    3981/3981: "Tile_X+0000399700_Y+0000654500.laz" ... Done.
```

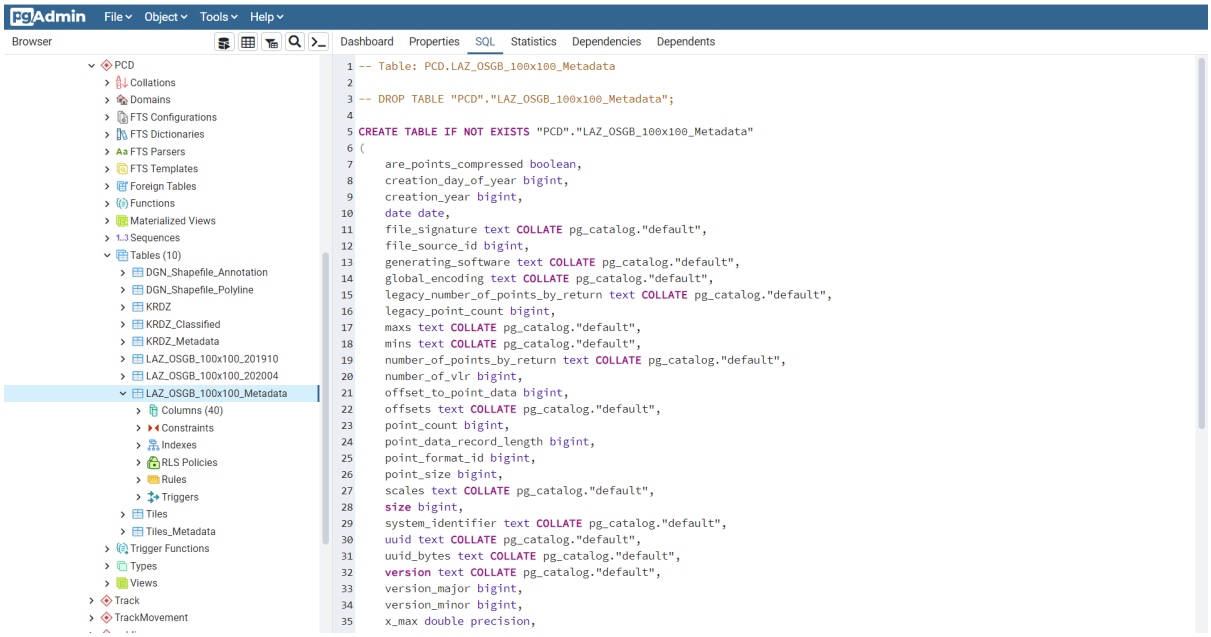


Figure 35: Snapshot of the "PCD"."LAZ_OSGB_100x100_Metadata" table.

PCD.import_laz_by_date

PCD.**import_laz_by_date**(*pcd_date*, *subset=None*, *incl_metadata=True*, *only_metadata=False*, *if_exists='fail'*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import LAZ data for a given date into the project database.

Parameters

- **pcd_date** (*str*) – Date of the point cloud data in the format 'YYYY-MM-DD'.
- **incl_metadata** (*bool*) – Whether to import the metadata along with the data; defaults to `True`.
- **only_metadata** (*bool*) – Whether to import only the metadata and not the data itself; defaults to `False`.
- **subset** (*list | tuple | numpy.ndarray | range | None*) – List of indexes indicating specific local file paths to import; defaults to `None`.

- **if_exists** (*str*) – Action to take if the targeted table already exists (see pandas.DataFrame.to_sql); defaults to `'fail'`.
- **confirmation_required** (*bool*) – Whether to ask for confirmation before proceeding; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Test (201910):

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> pcd.import_laz_by_date(pcd_date='201910', if_exists='append')
To import LAZ data (October 2019) into the table "PCD"."LAZ_OSGB_100x100_201910
↪"?
 [No]|Yes: yes
Importing the data ...
    1/1933: "Tile_X+0000374700_Y+0000673800.laz" ... Done.
    ...
    1933/1933: "Tile_X+0000399700_Y+0000654500.laz" ... Done.
```

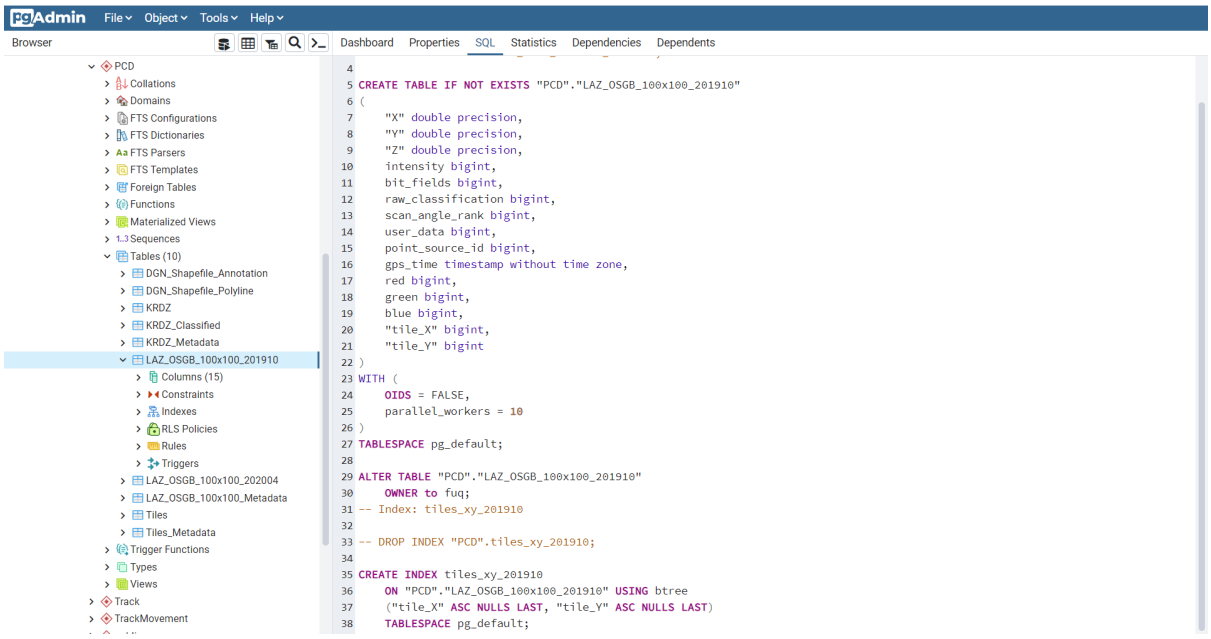


Figure 36: Snapshot of the "PCD"."LAZ_OSGB_100x100_201910" table.

Test (202004):

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> pcd.import_laz_by_date(pcd_date='202004', if_exists='append')
To import LAZ data (April 2020) into the table "PCD"."LAZ_OSGB_100x100_202004"?
 [No]|Yes: yes
Importing the data ...
```

(continues on next page)

(continued from previous page)

```
    1/2048: "Tile_X+0000340000_Y+0000674100.laz" ... Done.
    ...
    2048/2048: "Tile_X+0000399700_Y+0000654500.laz" ... Done.
```

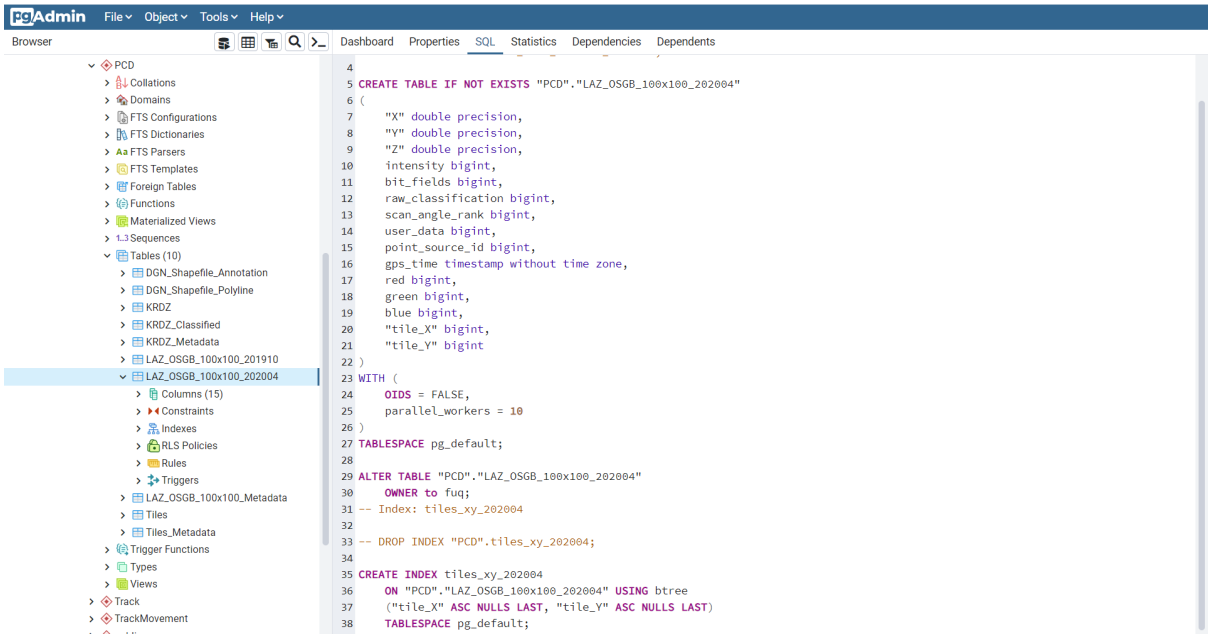


Figure 37: Snapshot of the "PCD"."LAZ_OSGB_100x100_202004" table.

Create an index:

```
# SET max_parallel_maintenance_workers TO 10;
SET max_parallel_workers TO 10;
SET maintenance_work_mem TO '10 GB';
SET checkpoint_timeout TO '180min';
SET max_wal_size TO '100GB';
SET min_wal_size TO '80MB';

ALTER TABLE "PCD"."LAZ_OSGB_100x100_202004" SET (parallel_workers = 10);

CREATE INDEX tiles_xy_202004 ON "PCD"."LAZ_OSGB_100x100_202004" ("tile_X",
 ↪"tile_Y");

SET enable_seqscan TO OFF;
```

Debugging (deprecated):

```
>>> # Check if there are missing files that weren't imported into the database
>>> from src.preprocessor import PCD
>>> from src.utils import TrackFixityDB
>>> from pyhelpers.dirs import cd
>>> import glob
>>> import natsort
>>> pcd = PCD()
>>> dat_date = '202004'
>>> db_instance = TrackFixityDB()
```

(continues on next page)

(continued from previous page)

```
>>> with db_instance.engine.connect() as conn:
...     query = (f"SELECT table_name FROM information_schema.tables "
...              f"WHERE table_schema='LAZ_OSGB_100x100_{dat_date}' "
...              f"AND table_type='BASE TABLE';")
...     res = conn.execute(sqlalchemy.text(query))
>>> temp_list = res.fetchall()
>>> table_list = [tn[0] for tn in temp_list if tn[0].startswith("Tile_X")]
>>> laz_dat_dir = cd(pcd.laz_dir_(dat_date))
>>> file_list = natsort.natsorted(glob.glob1(cd(laz_dat_dir), "*.laz"))
>>> file_list = [fn.replace(".laz", "") for fn in file_list]
>>> # A list of paths to the missing files
>>> missing_files = [cd(laz_dat_dir, x) for x in list(set(file_list) -␣
↪set(table_list))]
>>> missing_files
[]
```

PostgreSQL (PL/pgSQL) query for moving and renaming tables:

```
DO $$
DECLARE
    tbl_name TEXT;
BEGIN
    SET search_path='PointCloudData202004';
    FOR tbl_name IN
        SELECT table_name FROM information_schema.tables
        WHERE table_name like '%_202004_metadata'
            AND table_schema='PointCloudData202004'
            AND table_type='BASE TABLE'
        ORDER BY table_name
    LOOP
        EXECUTE FORMAT(
            'ALTER TABLE "%s" SET SCHEMA "PointCloudMetadata202004";', tbl_
↪name);
    END LOOP;
END; $$;

DO $$
DECLARE
    tbl_name TEXT;
BEGIN
    SET search_path='PointCloudMetadata202004';
    FOR tbl_name IN
        SELECT table_name FROM information_schema.tables
        WHERE table_name like '%_202004_metadata'
            AND table_schema='PointCloudMetadata202004'
            AND table_type='BASE TABLE'
        ORDER BY table_name
    LOOP
        EXECUTE FORMAT(
            'ALTER TABLE "%s" RENAME TO "%s";', tbl_name, SUBSTRING(tbl_name, 1,
↪ 30));
    END LOOP;
END; $$;


DO $$
```

(continues on next page)

(continued from previous page)

```
DECLARE
    tbl_name TEXT;
BEGIN
    SET search_path='PointCloudData202004';
    FOR tbl_name IN
        SELECT table_name FROM information_schema.tables
        WHERE table_name like '%_202004'
            AND table_schema='PointCloudData202004'
            AND table_type='BASE TABLE'
        ORDER BY table_name
    LOOP
        EXECUTE FORMAT(
            'ALTER TABLE "%s" RENAME TO "%s";', tbl_name, SUBSTRING(tbl_name, 1,
 ↪ 30));
    END LOOP;
END; $$;
```

PCD.import_tiles

PCD.**import_tiles**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import information about tiles into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data files; defaults to `False`.
- **confirmation_required** (*bool*) – Whether to ask for confirmation to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> pcd.import_tiles()
To import data of tiles into the schema "PCD"?
 [No]|Yes: yes
Importing the metadata ... Done.
Importing information about coordinates of the tiles ... Done.
```

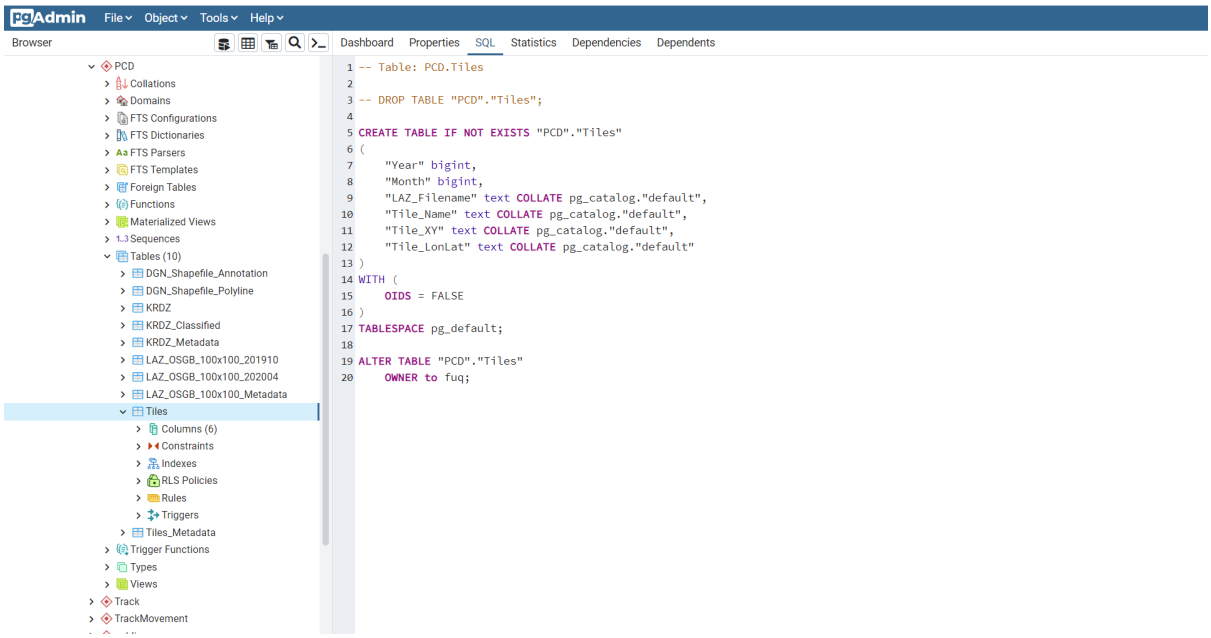


Figure 38: Snapshot of the "PCD"."Tiles" table.

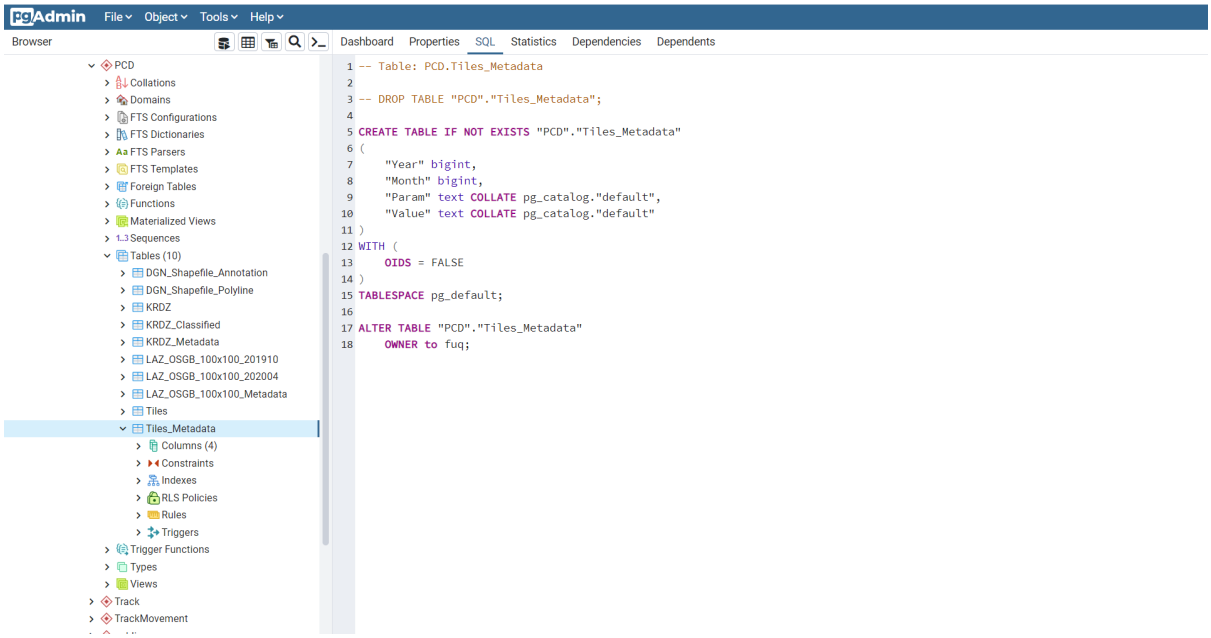


Figure 39: Snapshot of the "PCD"."Tiles_Metadata" table.

PCD.load_dgn_shp

PCD.**load_dgn_shp**(*pcd_date=None*, *layer_name=None*)

Load data of the DGN-converted shapefile from the project database.

Parameters

- **pcd_date** (*str | int | None*) – Date of the DGN data; defaults to None.
- **layer_name** (*str | None*) – Layer name of the DGN-converted shapefile; defaults to None.

Returns

Raw data of the DGN-converted shapefile.

Return type

tuple

Note:

The returned tuple contains two dictionaries:

- 'Annotation': The 'Annotation' layer.
- 'Polyline': The 'Complex Chain' and 'LineString' entities.

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> dgn_pl, dgn_pl_pcd = pcd.load_dgn_shp()
>>> type(dgn_pl)
dict
>>> list(dgn_pl.keys())
['Annotation', 'Polyline']
>>> dgn_pl['Annotation'].shape
(305249, 49)
>>> dgn_pl['Polyline'].shape
(615378, 28)
>>> type(dgn_pl_pcd)
dict
>>> list(dgn_pl_pcd.keys())
['Annotation', 'Polyline']
>>> dgn_pl_pcd['Annotation'].shape
(305249, 3)
>>> list(dgn_pl_pcd['Polyline'].keys())
['Complex Chain', 'LineString']
>>> # 'Polyline' data of April 2020
>>> dgn_pl, dgn_pl_pcd = pcd.load_dgn_shp(pcd_date='202004', layer_name=
↪'Polyline')
>>> dgn_pl.shape
(310118, 28)
>>> type(dgn_pl_pcd)
dict
```

(continues on next page)

(continued from previous page)

```
>>> list(dgn_pl_pcd.keys())
['Complex Chain', 'LineString']
>>> dgn_pl_pcd['Complex Chain'].shape
(1068396, 3)
>>> dgn_pl_pcd['LineString'].shape
(620208, 3)
>>> # 'Annotation' data of April 2020
>>> dgn_annot, dgn_annot_pcd = pcd.load_dgn_shp('202004', layer_name='Annotation
↪')
>>> dgn_annot.shape
(152628, 49)
>>> dgn_annot_pcd.shape
(152628, 3)
```

PCD.load_krdz

PCD.**load_krdz**(*pcd_dates=None*, ***kwargs*)

Load KRDZ data (for a given data date or dates) from the project database.

Parameters

- **pcd_dates** (*str | list | None*) – Date(s) of the point cloud data; defaults to None.
- **kwargs** – [Optional] additional parameters of the method pyhelpers.dbms.PostgreSQL.read_sql_query.

Returns

KRDZ data for the given date or dates.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> krdz_dat_201910 = pcd.load_krdz('201910')
>>> krdz_dat_201910.shape
(152621, 29)
>>> krdz_dat = pcd.load_krdz()
>>> krdz_dat.shape
(305249, 29)
```

PCD.load_laz

PCD.**load_laz**(*tile_xy*, *pcd_dates=None*, *greyscale=True*, *gs_coef=1.2*, *limit=None*, ***kwargs*)

Load LAZ data within a given tile and dates of the point cloud data from the project database.

Parameters

- **tile_xy** (*tuple | list*) – Easting (X) and northing (Y) coordinates of the tile.
- **pcd_dates** (*str | list | None*) – Date(s) of the point cloud data; defaults to `None`.
- **greyscale** (*bool*) – Whether to transform the colour data to greyscale; defaults to `True`.
- **gs_coef** (*float*) – Coefficient for adjusting intensity (only when `greyscale=True`); defaults to `1.2`.
- **limit** (*int*) – Limit on the number of rows to query from the database; defaults to `None`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.read_sql_query.

Returns

Dictionary containing XYZ and RGB data.

Return type

dict

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> xyz_rgb_dat = pcd.load_laz(tile_xy=(340500, 674200), limit=10000)
>>> type(xyz_rgb_dat)
dict
>>> list(xyz_rgb_dat.keys())
['201910', '202004']
>>> type(xyz_rgb_dat['201910'][0])
numpy.ndarray
>>> xyz_rgb_dat['201910'][0].shape
(10000, 3)
>>> type(xyz_rgb_dat['202004'][0])
numpy.ndarray
>>> xyz_rgb_dat['202004'][0].shape
(10000, 3)
```

PCD.load_tiles

PCD.**load_tiles**(*pcd_date=None*)

Load data of the tiles for point cloud data.

Parameters

pcd_date (*str | int | None*) – Date of the point cloud data in the format 'YYYY-MM-DD'; defaults to None.

Returns

Data of tiles.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> tiles_dat = pcd.load_tiles(pcd_date='201910')
>>> tiles_dat.head()
   Year  ...                                        Tile_LonLat
0  2019  ...  POLYGON ((-2.960910975517333 55.95617100494877...
1  2019  ...  POLYGON ((-2.960888715229412 55.95527265723062...
2  2019  ...  POLYGON ((-2.012726922542862 55.77482976892926...
3  2019  ...  POLYGON ((-2.009538521884446 55.77393156353337...
4  2019  ...  POLYGON ((-2.01113257284158 55.77393142614289,...
[5 rows x 6 columns]
>>> tiles_dat = pcd.load_tiles(pcd_date='202004')
>>> tiles_dat.head()
   Year  ...                                        Tile_LonLat
0  2020  ...  POLYGON ((-2.962512280825055 55.95615850596208...
1  2020  ...  POLYGON ((-2.960888715229412 55.95527265723062...
2  2020  ...  POLYGON ((-2.960910975517333 55.95617100494877...
3  2020  ...  POLYGON ((-2.960933237072112 55.95706935252924...
4  2020  ...  POLYGON ((-2.959287445726476 55.95528513499585...
[5 rows x 6 columns]
>>> tiles_dat = pcd.load_tiles()
>>> tiles_dat.shape
(3981, 6)
```

PCD.map_view_tiles

PCD.**map_view_tiles**(*tile_colours=None, update=False, verbose=True*)

Make a map view of tiles of all available point cloud data.

Parameters

- **tile_colours** (*list | None*) – List of hex colour codes for tiles on the map view; defaults to None.
- **update** (*bool*) – Whether to reprocess the original data files; defaults to False.

- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `False`.

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> pcd.map_view_tiles()
```

Figure 40: Tiles for the point cloud data.

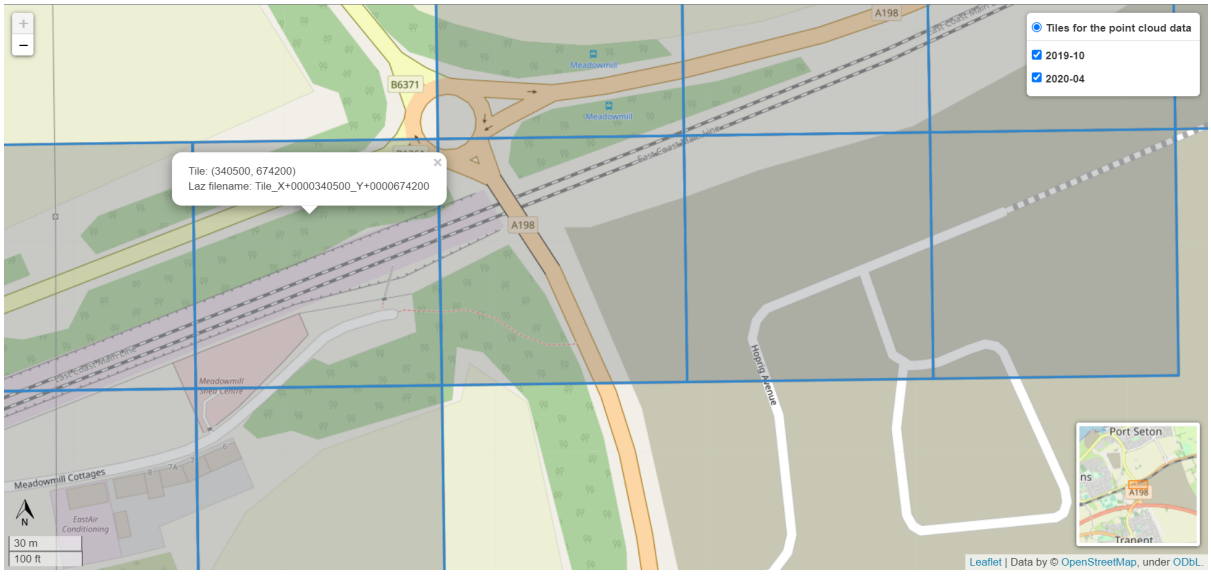


Figure 41: Tiles (zoomed in) for the point cloud data.

PCD.map_view_tiles_by_date

PCD.**map_view_tiles_by_date**(*pcd_date*, *tile_colour='#3186CC'*, *update=False*, *verbose=True*)

Make a map view of tiles of point cloud data for a given date.

Parameters

- **pcd_date** (*str*) – Date of the point cloud data in the format 'YYYY-MM-DD'.
- **tile_colour** (*str*) – Hex colour code for tiles on the map view; defaults to `'#3186CC'`.
- **update** (*bool*) – Whether to reprocess the original data files; defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `True`.

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
```

October 2019:

```
>>> pcd.map_view_tiles_by_date(pcd_date='201910', tile_colour='#DB7B2B')
```

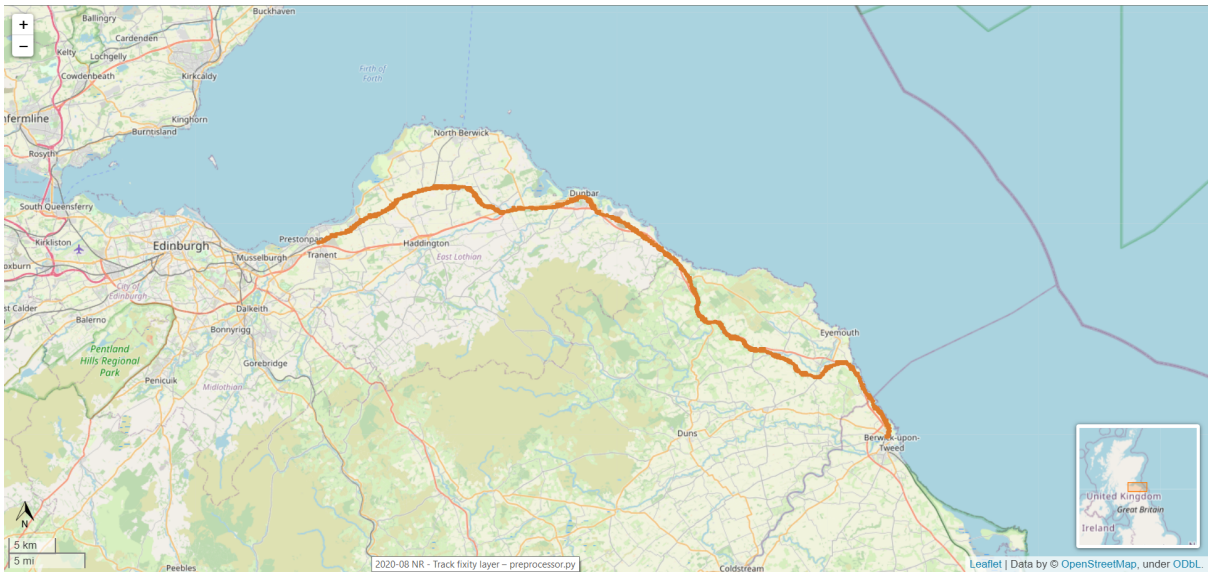


Figure 42: Tiles for the point cloud data (October 2019).

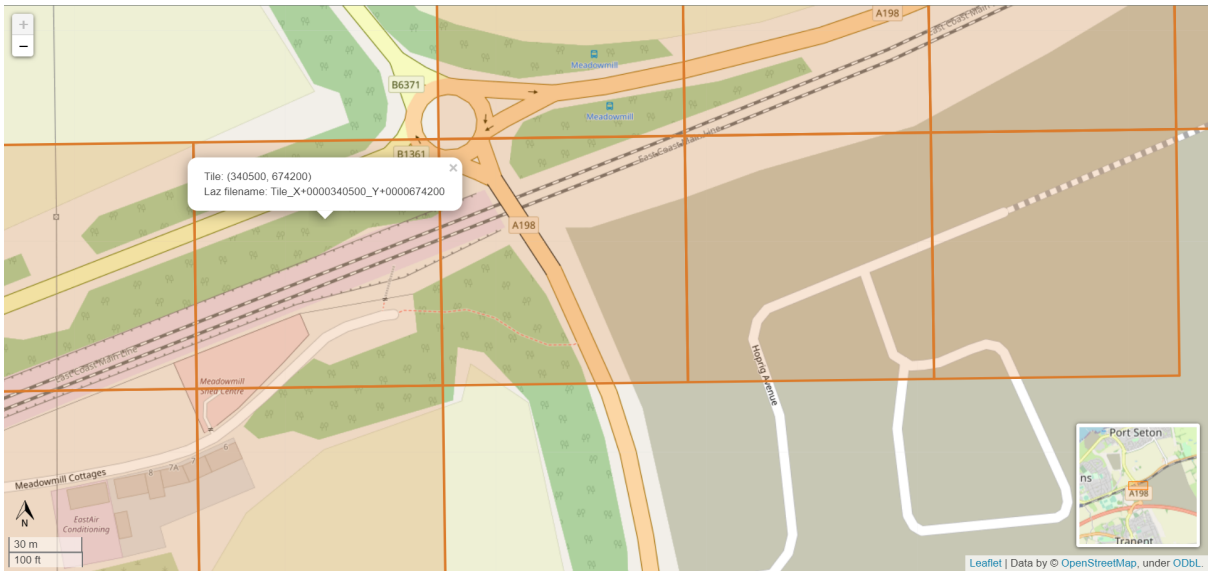


Figure 43: Tiles (zoomed in) for the point cloud data (October 2019).

April 2020:

```
>>> pcd.map_view_tiles_by_date(pcd_date='202004', tile_colour='#3186CC')
```

Figure 44: Tiles for the point cloud data (April 2020).

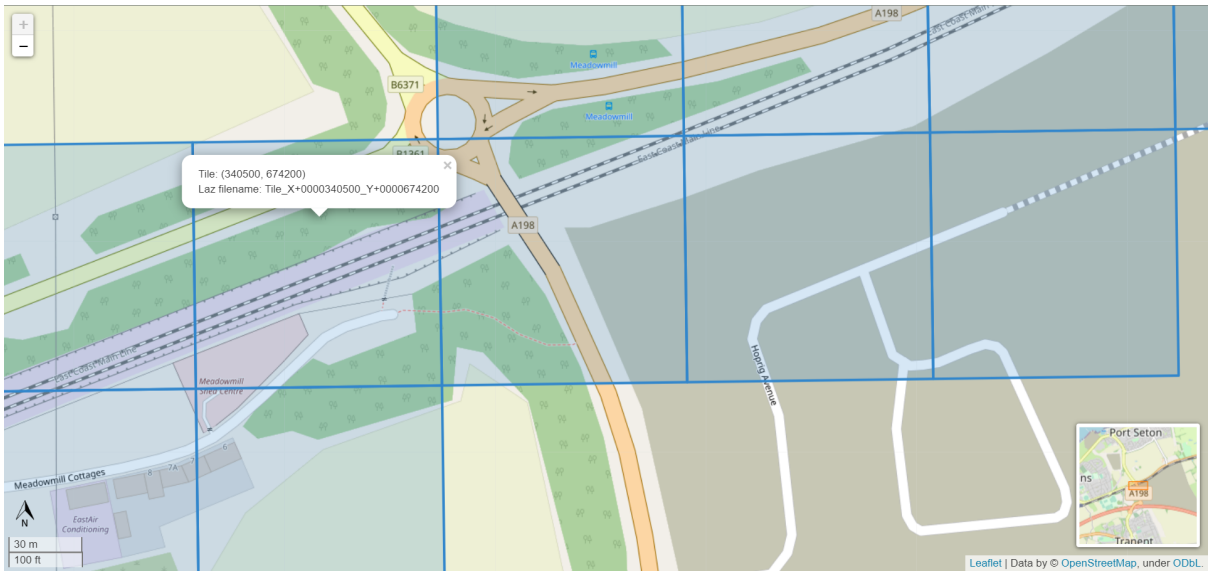


Figure 45: Tiles (zoomed in) for the point cloud data (April 2020).

PCD.parse_laz

PCD.**parse_laz**(*path_to_laz*, *header_to_dataframe=True*, *points_to_dataframe=True*, ***kwargs*)

Transform LAS/LAZ data into a dataframe.

Parameters

- **path_to_laz** (*str*) – Pathname of a LAZ/LAS file.
- **header_to_dataframe** (*bool*) – Whether to convert the header data to a dataframe; defaults to `True`.
- **points_to_dataframe** (*bool*) – Whether to convert the points data to a dataframe; defaults to `True`.
- **kwargs** – [Optional] additional parameters for pylas.read.

Returns

LAS/LAZ data, including header and points.

Return type

tuple

Examples:

```
>>> from src.preprocessor import PCD
>>> from pyhelpers.dirs import cd
>>> pcd = PCD()
>>> tile_x, tile_y = (340500, 674200)
>>> path_to_laz = cd(pcd.laz_dir_('201910'), f"Tile_X+0000{tile_x}_Y+0000{tile_
↪y}.laz")
>>> header_data, laz_points_data = pcd.parse_laz(path_to_laz)
>>> header_data
```

(continues on next page)

(continued from previous page)

```
                                DEFAULT_POINT_FORMAT  ... z_scale
0  {'dimension_names': <generator object PointFor...  ...   0.001
[1 rows x 39 columns]
>>> laz_points_data.head()
                   X           Y       Z  ...  red  green  blue
0         340600.000  674239.695  41.676  ...  255    255   255
1         340600.000  674240.305  40.457  ...  255    255   255
2         340600.000  674239.609  41.880  ...  255    255   255
3         340600.000  674238.253  41.249  ...  255    255   255
4         340600.000  674238.152  39.933  ...  255    255   255
[5 rows x 13 columns]
>>> header_data, laz_points_data = pcd.parse_laz(
...     path_to_laz, header_to_dataframe=False, points_to_dataframe=False)
>>> type(header_data)
dict
>>> list(header_data.keys())[:5]
['DEFAULT_POINT_FORMAT',
 'DEFAULT_VERSION',
 'are_points_compressed',
 'creation_date',
 'evlrs']
>>> type(laz_points_data)
numpy.ndarray
>>> laz_points_data.shape
(54007138, 13)
```

PCD.read_dgn_shp

PCD.**read_dgn_shp**(*update=False*, *verbose=False*, ***kwargs*)

Read all available DGN-converted shapefiles from a local directory.

Parameters

- **update** (*bool*) – Whether to re-convert and read the original DGN data; defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.
- **kwargs** – [Optional] additional parameters for the function `dgn2shp()`.

Returns

Data of DGN-converted shapefiles or `None` if no files are found.

Return type

dict | None

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> dgn_shp = pcd.read_dgn_shp()
>>> type(dgn_shp)
```

(continues on next page)

(continued from previous page)

```
dict
>>> list(dgn_shp.keys())
['Annotation', 'MultiPatch', 'Polygon', 'Polyline', 'Point']
>>> dgn_shp['Annotation'].shape
(305249, 49)
>>> dgn_shp['Annotation'].head()
        Year  Month  ... TxtMemo                                  geometry
0       2019     10  ...       1  POINT Z (340131.735 674089.424 33.353)
1       2019     10  ...       2  POINT Z (340132.204 674089.599 33.352)
2       2019     10  ...       3  POINT Z (340133.141 674089.947 33.348)
3       2019     10  ...       4  POINT Z (340134.078 674090.296 33.344)
4       2019     10  ...       5  POINT Z (340135.015 674090.645 33.340)
[5 rows x 49 columns]
>>> dgn_shp['Polyline'].shape
(615378, 28)
>>> dgn_shp['Polyline'].head()
        Year  Month  ... QrotZ                                          ␣
 ↪geometry
0       2019     10  ...   0.0  LINESTRING Z (340131.735 674089.424 33.343, 34..
 ↪.
1       2019     10  ...   0.0  LINESTRING Z (340132.204 674089.599 33.342, 34..
 ↪.
2       2019     10  ...   0.0  LINESTRING Z (340133.141 674089.947 33.338, 34..
 ↪.
3       2019     10  ...   0.0  LINESTRING Z (340134.078 674090.296 33.334, 34..
 ↪.
4       2019     10  ...   0.0  LINESTRING Z (340135.015 674090.645 33.330, 34..
 ↪.
[5 rows x 28 columns]
```

PCD.read_dgn_shp_by_date

PCD.**read_dgn_shp_by_date**(*pcd_date=None*, *update=False*, *confirmation_required=True*, *verbose=False*, *rm_dgn_shp_files=False*, ***kwargs*)

Read DGN-converted shapefiles for a specific date of the point cloud data.

Parameters

- **pcd_date** (*str | None*) – Date of the point cloud data, e.g., `'201910'`, `'202004'`. When `pcd_date=None` (default), the function reads shapefiles for all available "ECM8_10_66_*_20200625" DGN data.
- **update** (*bool*) – Whether to re-convert and read the original DGN data; defaults to `False`.
- **confirmation_required** (*bool*) – Whether to ask for confirmation to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.
- **rm_dgn_shp_files** (*bool*) – Whether to remove the converted shapefiles; defaults to `False`.

- **kwargs** – [Optional] additional parameters for the function `dgn2shp()`.

Returns
Shapefile data of "ECM8_10_66_*_20200625".

Return type
dict

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> dgn_shp_files = pcd.read_dgn_shp_by_date()
>>> type(dgn_shp_files)
dict
>>> list(dgn_shp_files.keys())
['201910', '202004']
>>> dgn_shp_data_202004 = dgn_shp_files['202004']
>>> type(dgn_shp_data_202004)
dict
>>> list(dgn_shp_data_202004.keys())
['Annotation', 'MultiPatch', 'Point', 'Polygon', 'Polyline']
>>> dgn_shp_data_202004['Annotation'].shape
(152628, 47)
>>> dgn_shp_data_202004['Annotation'].head()
      Entity  ...                                   geometry
0       Text  ...  POINT Z (340131.462 674089.321 33.355)
1       Text  ...  POINT Z (340131.931 674089.495 33.353)
2       Text  ...  POINT Z (340132.868 674089.844 33.350)
3       Text  ...  POINT Z (340133.805 674090.192 33.345)
4       Text  ...  POINT Z (340134.742 674090.542 33.341)
[5 rows x 47 columns]
>>> dgn_shp_data_202004['Polyline'].shape
(310118, 26)
>>> dgn_shp_data_202004['Polyline'].head()
              Entity  ...                                           geometry
0         LineString  ...  LINESTRING Z (340131.462 674089.321 33.345, 34...
1         LineString  ...  LINESTRING Z (340131.931 674089.495 33.343, 34...
2         LineString  ...  LINESTRING Z (340132.868 674089.844 33.340, 34...
3         LineString  ...  LINESTRING Z (340133.805 674090.192 33.335, 34...
4         LineString  ...  LINESTRING Z (340134.742 674090.542 33.331, 34...
[5 rows x 26 columns]
```

PCD.read_krdz

PCD.**read_krdz**(*update=False*, *verbose=False*)

Read a KRDZ file from a local directory.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Returns
Data from the KRDZ file as a pandas DataFrame, or `None` if no data is found.

Return type
pandas.DataFrame | None

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> krdz_file = pcd.read_krdz()
>>> krdz_file.shape
(305249, 29)
```

PCD.read_krdz_by_date

PCD.**read_krdz_by_date**(*pcd_date*, *update=False*, *verbose=False*)

Read a KRDZ file for a given date from a local directory.

Parameters

- **pcd_date** (*str*) – Date of the KRDZ file in the format 'YYYYMMDD'.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Returns
Data from the KRDZ file as a pandas DataFrame, or `None` if no data is found.

Return type
pandas.DataFrame | None

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> krdz_file_201910 = pcd.read_krdz_by_date(pcd_date='201910')
>>> krdz_file_201910.shape
(152621, 27)
>>> krdz_file_202004 = pcd.read_krdz_by_date(pcd_date='202004')
>>> krdz_file_202004.shape
(152628, 27)
```

PCD.read_krdz_metadata

PCD.**read_krdz_metadata**(*update=False*, *verbose=False*)

Read the metadata for the KRDZ data from a local directory.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.

Returns

Column names and descriptions of the KRDZ data.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> krdz_schema_info = pcd.read_krdz_metadata()
>>> krdz_schema_info.shape
(27, 2)
```

PCD.read_tiles_prj

PCD.**read_tiles_prj**(*update=False*, *verbose=False*)

Read all PRJ files for tiles from a local directory.

Parameters

- **update** (*bool*) – Whether to reprocess the original data files; defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `False`.

Returns

Metadata and information of each tile as a dictionary.

Return type

dict

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
>>> project_prj = pcd.read_tiles_prj()
>>> type(project_prj)
dict
>>> list(project_prj.keys())
['Metadata', 'Projection']
```

(continues on next page)

(continued from previous page)

```
>>> project_prj['Metadata'].shape
(24, 4)
>>> project_prj['Projection'].shape
(3981, 6)
```

PCD.read_tiles_prj_by_date

PCD.**read_tiles_prj_by_date**(*pcd_date*, *update=False*, *verbose=False*)

Read a PRJ file of tiles for a given date from a local directory.

Parameters

- **pcd_date** (*str*) – Date of the point cloud data in the format 'YYYY-MM-DD'.
- **update** (*bool*) – Whether to reprocess the original data files; defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `False`.

Returns

Metadata and information of each tile.

Return type

dict | None

Examples:

```
>>> from src.preprocessor import PCD
>>> pcd = PCD()
```

October 2019:

```
>>> project_prj = pcd.read_tiles_prj_by_date(pcd_date='201910')
>>> type(project_prj)
dict
>>> list(project_prj.keys())
['Metadata', 'Projection']
>>> project_prj['Metadata'].shape
(12, 2)
>>> project_prj['Projection'].shape
(1933, 4)
```

April 2020:

```
>>> project_prj = pcd.read_tiles_prj_by_date(pcd_date='202004')
>>> type(project_prj)
dict
>>> list(project_prj.keys())
['Metadata', 'Projection']
>>> project_prj['Metadata'].shape
(12, 2)
```

(continues on next page)

(continued from previous page)

```
>>> project_prj['Projection'].shape
(2048, 4)
```

2.2.9 Reports

class src.preprocessor.**Reports**(*db_instance=None*)

Spreadsheet-based reports.

Note: The data handled by this class included a number of spreadsheet reports up to 2020-08-22, regarding the rail tracks. These data were not yet explored for this project.

Parameters

db_instance (TrackFixityDB | *None*) – PostgreSQL database instance; defaults to None.

Variables

- **data_filenames** (*list*) – Filenames of all the original data files.
- **data_pathnames** (*list*) – Pathnames of all the original data files.
- **mileage_dtypes** (*dict*) – Data type of mileages.
- **db_instance** (TrackFixityDB) – PostgreSQL database instance.

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> reports.NAME
'Reports'
```

Attributes:

DATA_DIR	Pathname of a local directory where the data is stored.
DEFRAG_REPORT_FILENAME	Filename of the data of defrag report.
ETM_FILENAME	Filename of the data of equated track miles.
INM_CDR_DST_FILENAME	Filename of the data of INM combined report for DST.
INM_CDR_FILENAME	Filename of the data of INM combined data report.
JCT_FILENAME	Filename of the data of junctions.
NAME	Data name.
PLSAO_FILENAME	Filename of the data of plain line SnC attributes overlap.
PLSAQ_FILENAME	Filename of the data of plain line SnC attributes gap.
RGR_FILENAME	Filename of the data of responsibility gap report.
TAGR_FILENAME	Filename of the data of track attributes gap report.
TCGR_FILENAME	Filename of the data of track category gap report.
TCWRS_FILENAME	Filename of the data of track category with responsibility and switch.
TCWR_FILENAME	Filename of the data of track category with responsibility.
TRK_REPORT_FILENAME	Filename of the data of track report.

Reports.DATA_DIR

Reports.**DATA_DIR: str = 'data\\Reports\\20200822'**

Pathname of a local directory where the data is stored.

Reports.DEFRAG_REPORT_FILENAME

Reports.**DEFRAG_REPORT_FILENAME: str = 'Defrag_report.zip'**

Filename of the data of defrag report.

Reports.ETM_FILENAME

Reports.**ETM_FILENAME: str = 'Equated_track_miles.zip'**

Filename of the data of equated track miles.

Reports.INM_CDR_DST_FILENAME

Reports.**INM_CDR_DST_FILENAME: str = 'INM_combined_report_for_DST.zip'**

Filename of the data of INM combined report for DST.

Reports.INM_CDR_FILENAME

Reports.**INM_CDR_FILENAME: str = 'INM_combined_data_report.zip'**

Filename of the data of INM combined data report.

Reports.JCT_FILENAME

Reports.**JCT_FILENAME: str = 'Junctions.zip'**

Filename of the data of junctions.

Reports.NAME

Reports.**NAME: str = 'Reports'**

Data name.

Reports.PLSAO_FILENAME

Reports.**PLSAO_FILENAME: str = 'Plain_line_SnC_attributes_overlap.zip'**

Filename of the data of plain line SnC attributes overlap.

Reports.PLSAQ_FILENAME

Reports.**PLSAQ_FILENAME: str = 'Plain_line_SnC_attributes_gap.zip'**

Filename of the data of plain line SnC attributes gap.

Reports.RGR_FILENAME

Reports.**RGR_FILENAME: str = 'Responsibility_gap_report.zip'**

Filename of the data of responsibility gap report.

Reports.TAGR_FILENAME

Reports.**TAGR_FILENAME: str = 'Track_attributes_gap_report.zip'**

Filename of the data of track attributes gap report.

Reports.TCGR_FILENAME

Reports.**TCGR_FILENAME: str = 'Track_category_gap_report.zip'**

Filename of the data of track category gap report.

Reports.TCWRS_FILENAME

Reports.**TCWRS_FILENAME: str =**
'Track_category_with_responsibility_and_switch.zip'

Filename of the data of track category with responsibility and switch.

Reports.TCWR_FILENAME

Reports.**TCWR_FILENAME: str = 'Track_category_with_responsibility.zip'**

Filename of the data of track category with responsibility.

Reports.TRK_REPORT_FILENAME

Reports.**TRK_REPORT_FILENAME: str = '9999_Track_report.zip'**

Filename of the data of track report.

Methods:

cdd(*filename[, mkdir])	Change directory to the report pack.
read_defrag_report([update, verbose])	Read data of defrag report.
read_etm([update, verbose])	Read data of equated track miles.
read_inm_cdr([update, verbose])	Read INM combined report data.
read_inm_dst([update, verbose])	Read INM combined report data for DST (Decision Support Tool).
read_junctions([update, verbose])	Read data of junctions.
read_plsag([update, verbose])	Read data of plain line SnC attributes gap.
read_plsao([update, verbose])	Read data of plain line SnC attributes overlap.
read_rgr([update, verbose])	Read report data of responsibility gap.
read_tagr([update, verbose])	Read report data of track attributes gap.
read_tcgr([update, verbose])	Read report data of track category gap report.
read_tcr([update, verbose])	Read data of track category with responsibility.
read_tcrs([update, verbose])	Read data of track category with responsibility and switch.
read_track_report([update, verbose])	Read data of track report.

Reports.cdd

Reports.**cdd**(**filename*, *mkdir=True*)

Change directory to the report pack.

Parameters

- **filename** (*str*) – Name of a file.
- **mkdir** (*bool*) – Whether to create a subdirectory (if it does not exist); defaults to True.

Returns

Pathname of the local directory of the report pack (and subdirectories and/or files within it).

Return type

str

Reports.read_defrag_report

Reports.**read_defrag_report**(*update=False*, *verbose=False*)

Read data of defrag report.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns

Data of defrag report.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # defrag_report = reports.read_defrag_report(update=True, verbose=True)
>>> defrag_report = reports.read_defrag_report()
>>> defrag_report.shape
(529176, 71)
```

Note:

- 'REF_TRACKID' (int): 0 and nan
- 'SMOOTH_TRACK_CAT' (int, 1-6): nan and '1A'
- 'TRACK_CATEGORY' (int, 1-6): nan, 'Track Category without value' and '1A'
- 'SPEED_LEVEL_RAISED': nan, 'Track Category without value'

Reports.read_etm

Reports.**read_etm**(*update=False*, *verbose=False*)

Read data of equated track miles.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns

Data of equated track miles.

Return type
pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # equated_track_miles = reports.read_etm(update=True, verbose=True)
>>> equated_track_miles = reports.read_etm()
>>> equated_track_miles.shape
(37953, 8)
```

Reports.read_inm_cdr

Reports.**read_inm_cdr**(*update=False*, *verbose=False*)

Read INM combined report data.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns
INM combined report data.

Return type
pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # inm_cdr = reports.read_inm_cdr(update=True, verbose=True)
>>> inm_cdr = reports.read_inm_cdr()
>>> inm_cdr.shape
(487187, 55)
```

Note:

- 'SLEEPERSPER60FOOTLENGTH' (int): strings (i.e. 'PAN' and 'NONE') and nan
- 'PATCHPROGRAM' (str of 2 digits of int): 2019, 'Patch Program' and nan
- 'TRACK_CATEGORY' (int, 1-6): nan and '1A'
- 'SMOOTH_TRACK_CAT' (int, 1-6): nan and '1A'

Reports.read_inm_dst

Reports.**read_inm_dst**(*update=False*, *verbose=False*)

Read INM combined report data for DST (Decision Support Tool).

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns

INM combined report data for DST.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # inm_dst = reports.read_inm_dst(update=True, verbose=True)
>>> inm_dst = reports.read_inm_dst()
>>> inm_dst.shape
(489973, 60)
```

Note:

- 'SLEEPERSPER60FOOTLENGTH' (int): nan, 'N', 'PAN', 'NONE' and 'AoR Rec without value'
- 'PATCHPROGRAM' (str of 2 digits of int): nan, 2019, 'Patch Program' and 'AoR Rec without value'
- 'TRACK_CATEGORY' (int, 1-6): nan, '1A' and 'Track Category without value'
- 'SMOOTH_TRACK_CAT' (int, 1-6): nan and '1A'
- 'SPEED_LEVEL_RAISED': nan and 'Track Category without value'

Reports.read_junctions

Reports.**read_junctions**(*update=False*, *verbose=False*)

Read data of junctions.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns
Data of junctions.

Return type
pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # junctions = reports.read_junctions(update=True, verbose=True)
>>> junctions = reports.read_junctions()
>>> junctions.shape
(3990, 8)
```

Note: from pyrcs.utils import nr_mileage_to_yards

end_m = junctions_csv.ELR_ENDMEASURE[0] start_m
= junctions_csv.ELR_STARTMEASURE[0] total_m_j
= junctions_csv.TOTALMILEGE[0] total_m_sandc =
junctions_csv.TOTALMILEGE[1]

total_yards = nr_mileage_to_yards(end_m) - nr_mileage_to_yards(start_m)

total_mi = total_m_j + total_m_sandc total_m_yards =
round(measurement.measures.Distance(mi=total_mi).yd)

total_yards == total_m_yards # True

Reports.read_plsag

Reports.**read_plsag**(*update=False*, *verbose=False*)

Read data of plain line SnC attributes gap.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns
Data of plain line SnC attributes gap.

Return type
pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # plsag = reports.read_plsag(update=True, verbose=True)
```

(continues on next page)

(continued from previous page)

```
>>> plsag = reports.read_plsag()
>>> plsag.shape
(222804, 19)
```

Reports.read_plsao

Reports.**read_plsao**(*update=False*, *verbose=False*)

Read data of plain line SnC attributes overlap.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns

Data of plain line SnC attributes overlap.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # plsao = reports.read_plsao(update=True, verbose=True)
>>> plsao = reports.read_plsao()
>>> plsao.shape
(163725, 13)
```

Reports.read_rgr

Reports.**read_rgr**(*update=False*, *verbose=False*)

Read report data of responsibility gap.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns

Data of responsibility gap.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # responsibility_gap = reports.read_rgr(update=True, verbose=True)
>>> responsibility_gap = reports.read_rgr()
>>> responsibility_gap.shape
(56170, 14)
```

Reports.read_tagr

Reports.**read_tagr**(*update=False*, *verbose=False*)

Read report data of track attributes gap.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns

Data of track attributes gap.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # tagr = reports.read_tagr(update=True, verbose=True)
>>> tagr = reports.read_tagr()
>>> tagr.shape
(303327, 38)
```

Reports.read_tcgr

Reports.**read_tcgr**(*update=False*, *verbose=False*)

Read report data of track category gap report.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns

Data of track category gap report.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # tcgr = reports.read_tcgr(update=True, verbose=True)
>>> tcgr = reports.read_tcgr()
>>> tcgr.shape
(253202, 16)
```

Reports.read_tcr

Reports.**read_tcr**(*update=False*, *verbose=False*)

Read data of track category with responsibility.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns

Data of track category with responsibility.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # tcr_ = reports.read_tcr(update=True, verbose=True)
>>> tcr_ = reports.read_tcr()
>>> tcr_.shape
(251512, 24)
```

Reports.read_tcrs

Reports.**read_tcrs**(*update=False*, *verbose=False*)

Read data of track category with responsibility and switch.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to False.

Returns

Data of track category with responsibility and switch.

Return type
pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # tcrs = reports.read_tcrs(update=True, verbose=True)
>>> tcrs = reports.read_tcrs()
>>> tcrs.shape
(320799, 21)
```

Note: `'UNIQUE_ID'` (`int`): `nan` and `'NOTINGEOGIS'`

Reports.read_track_report

`Reports.`**`read_track_report`**(*update=False*, *verbose=False*)

Read data of track report.

Parameters

- **`update`** (*`bool`*) – Whether to reprocess the original data file(s); defaults to `False`.
- **`verbose`** (*`bool | int`*) – Whether to print relevant information to the console; defaults to `False`.

Returns
Data of track report.

Return type
pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Reports
>>> reports = Reports()
>>> # track_report = reports.read_track_report(update=True, verbose=True)
>>> track_report = reports.read_track_report()
>>> track_report.shape
(325, 13)
```

2.2.10 Track

`class` `src.preprocessor.`**`Track`**(*db_instance=None*)

Track layout and quality.

This class currently handles:

- (calibrated) shapefile of the track layout (including reference lines and changes)
- track quality files

Parameters

db_instance (TrackFixityDB / *None*) – PostgreSQL database instance; defaults to None.

Variables

- **shp_data_dates** (*list*) – Dates of the track shapefiles.
- **track_quality_tid** (*list*) – Track IDs for the track quality files.
- **dtf_pathnames** (*list*) – File paths to the track quality files.
- **dtf_data_dates** (*list*) – Dates of the track quality files.
- **db_instance** (*pyhelpers.dbms.PostgreSQL*) – PostgreSQL database instance.

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> trk.NAME
'Track'
```

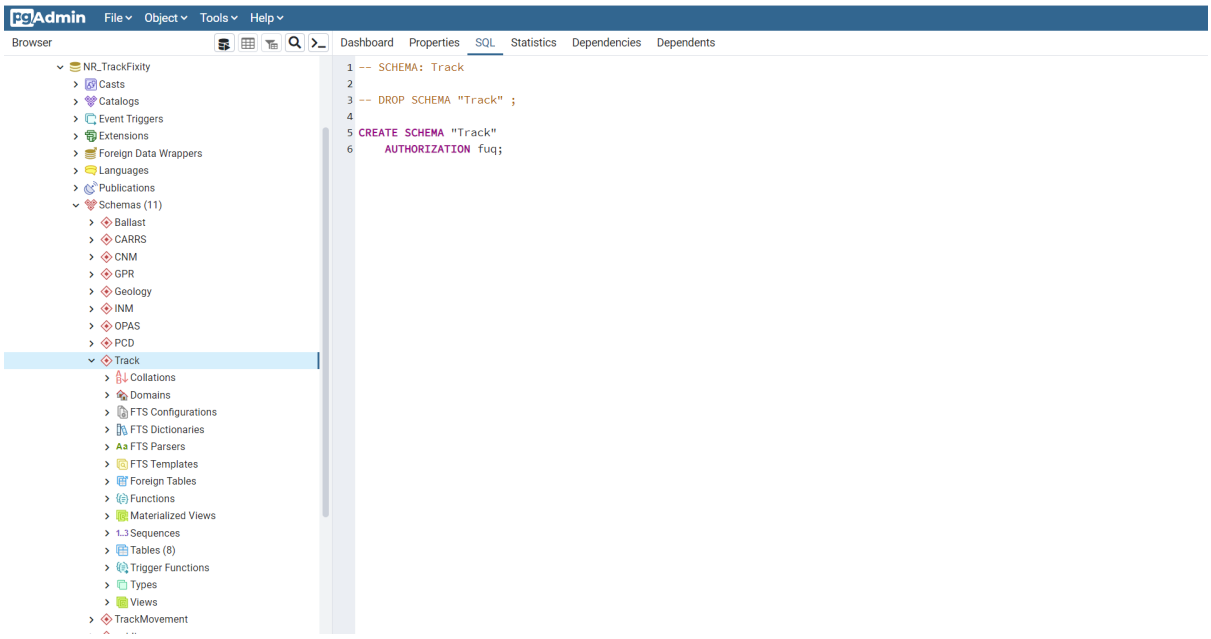


Figure 46: Snapshot of the *Track* schema.

Attributes:

DATA_DIR	Pathname of the local directory where the track data are stored.
DTF_DIR_PATH	Pathname of the local directory where the track quality files are stored.
DTF_KEY	Key of the parsed track quality data (in `dict` format).
DTF_TABLE_NAME	Name of the table storing the track quality files.
NAME	Name of the data.
NM_CHANGES_DESC	Short description of Network Model gauging changes data.
NM_CHANGES_FILENAME	Filename of Network Model gauging changes data.
PSEUDO_MILEAGE_TABLE_NAME	Name of the table storing pseudo mileages.
REF_LINE_DESC	Short description of reference line shapefile (calibrated) data.
REF_LINE_FILENAME	Filename of reference line shapefile (calibrated) data.
REF_LINE_TABLE_NAME	Name of the table storing reference line shapefile (calibrated) data.
SCHEMA_NAME	Name of the schema for storing the track data.
SHP_DIR_PATH	Pathname of the local directory where the track shapefiles are stored.
TRK_SHP_DESC	Short description of track shapefile (calibrated) data.
TRK_SHP_FILENAME	Filename of track shapefile (calibrated) data.
TRK_SHP_TABLE_NAME	Name of the table storing the track shapefile (calibrated) data.

Track.DATA_DIR

Track.**DATA_DIR: str = 'data\\Track'**

Pathname of the local directory where the track data are stored.

Track.DTF_DIR_PATH

Track.**DTF_DIR_PATH: str = 'data\\Track\\Track quality'**

Pathname of the local directory where the track quality files are stored.

Track.DTF_KEY

Track.**DTF_KEY: str = 'TQ'**

Key of the parsed track quality data (in `dict` format).

Track.DTF_TABLE_NAME

Track.**DTF_TABLE_NAME: str = 'Track_quality'**

Name of the table storing the track quality files.

Track.NAME

Track.**NAME: str = 'Track'**

Name of the data.

Track.NM_CHANGES_DESC

Track.**NM_CHANGES_DESC: str = 'Network Model gauging changes (39pt1 to 39pt2)'**

Short description of Network Model gauging changes data.

Track.NM_CHANGES_FILENAME

Track.**NM_CHANGES_FILENAME: str = 'NM_changes_39pt1_to_39pt2_gauging'**

Filename of Network Model gauging changes data.

Track.PSEUDO_MILEAGE_TABLE_NAME

Track.**PSEUDO_MILEAGE_TABLE_NAME: str = 'Pseudo_mileage'**

Name of the table storing pseudo mileages.

Track.REF_LINE_DESC

Track.**REF_LINE_DESC: str = 'Shapefile (calibrated) of reference line'**

Short description of reference line shapefile (calibrated) data.

Track.REF_LINE_FILENAME

Track.**REF_LINE_FILENAME: str = 'Calibrated_ReferenceLine'**

Filename of reference line shapefile (calibrated) data.

Track.REF_LINE_TABLE_NAME

Track.**REF_LINE_TABLE_NAME: str = 'RefLine_shapefile_calibrated'**

Name of the table storing reference line shapefile (calibrated) data.

Track.SCHEMA_NAME

Track.**SCHEMA_NAME: str = 'Track'**

Name of the schema for storing the track data.

Track.SHP_DIR_PATH

Track.**SHP_DIR_PATH: str = 'data\\Track\\Shapefiles'**

Pathname of the local directory where the track shapefiles are stored.

Track.TRK_SHP_DESC

Track.**TRK_SHP_DESC: str = 'Shapefile (calibrated) of tracks of the whole UK'**

Short description of track shapefile (calibrated) data.

Track.TRK_SHP_FILENAME

Track.**TRK_SHP_FILENAME: str = 'tracks_whole_uk_April2020_Calibrated'**

Filename of track shapefile (calibrated) data.

Track.TRK_SHP_TABLE_NAME

Track.**TRK_SHP_TABLE_NAME: str = 'Tracks_shapefile_calibrated'**

Name of the table storing the track shapefile (calibrated) data.

Methods:

fetch_tracks_shapefiles(trk_date[, update, ...])	Fetch data of track shapefiles (calibrated) from a local directory.
fmt_dtf_file_date(tq_date)	Reformat the date (in the filename) of a track quality file.
get_dtf_pathname(tid, tq_date)	Get the path to a track quality file of a given track ID and date.
import_dtf(tid, tq_date[, update, ...])	Import data of a track quality file (of a given track ID and date) into the project database.
import_gauging_changes(trk_date[, update, ...])	Import the shapefile (calibrated) data of UK tracks into the project database.
import_pseudo_mileage_dict([elr, set_index, ...])	Import data of (pseudo) mileages into the project database.
import_ref_line_shp(trk_date[, update, ...])	Import the shapefile (calibrated) of the reference line into the project database.
import_track_quality([update, ...])	Import data of all track quality files into the project database.
import_tracks_shapefiles(trk_date[, update, ...])	Import data of track shapefiles into the project database.
import_tracks_shp(trk_date[, update, ...])	Import the shapefile (calibrated) data of UK tracks into the project database.
load_pseudo_mileage_dict([elr])	Load the data of (pseudo) mileages from the project database.
load_track_quality([elr, tq_date])	Load the data of track quality files from the project database.
load_tracks_shp([trk_date, elr])	Load the shapefile (calibrated) data of UK tracks from the project database.
make_pseudo_mileage_dict([elr, set_index])	Make a dictionary of (pseudo) mileages for all available track IDs based on track shapefiles.
parse_dtf(path_to_dtf)	Parse a track quality file (.dtf format).
read_dtf(tid, tq_date[, update, verbose])	Read track quality files (.dtf) for a given pair of track ID and date from a local directory.
read_gauging_changes(trk_date[, update, verbose])	Read data of network model changes on '39pt1_to_39pt2_gauging' from a local directory.
read_ref_line_shp(trk_date[, update, verbose])	Read the shapefile (calibrated) of the reference line from a local directory.
read_track_quality([update, verbose])	Read data of all available track quality files from a local directory.
read_tracks_shp(trk_date[, update, verbose])	Read the shapefile (calibrated) data of tracks of the whole UK from a local directory.

Track.fetch_tracks_shapefiles

Track.**fetch_tracks_shapefiles**(*trk_date*, *update=False*, *verbose=False*, ***kwargs*)

Fetch data of track shapefiles (calibrated) from a local directory.

Parameters

- **trk_date** (*str*) – Data date of track shapefiles.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.
- **kwargs** – [Optional] additional parameters for the method pydriosm.reader.SHPReadParse.read_shp.

Returns

Data of the track shapefiles.

Return type

dict | None

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> # trk_shp_data = trk.fetch_tracks_shapefiles('202004', update=True,␣
↪verbose=True)
>>> trk_shp_data = trk.fetch_tracks_shapefiles(trk_date='202004')
>>> type(trk_shp_data)
dict
>>> list(trk_shp_data.keys())
['UK Tracks', 'Network Model Changes', 'Reference Line']
>>> trk_shp_data['UK Tracks'].shape
(49667, 11)
>>> type(trk_shp_data['Network Model Changes'])
dict
>>> list(trk_shp_data['Network Model Changes'].keys())
['Links_added', 'Links_deleted', 'Nodes_added', 'Nodes_deleted']
>>> trk_shp_data['Network Model Changes']['Links_added'].shape
(3, 11)
>>> trk_shp_data['Network Model Changes']['Links_deleted'].shape
(17, 11)
>>> trk_shp_data['Network Model Changes']['Nodes_added'].shape
(2, 4)
>>> trk_shp_data['Network Model Changes']['Nodes_deleted'].shape
(13, 4)
>>> trk_shp_data['Reference Line'].shape
(1583, 22)
```

Track.fmt_dtf_file_date

static Track.**fmt_dtf_file_date**(*tq_date*)

Reformat the date (in the filename) of a track quality file.

Parameters
tq_date (*str*) – Date (in the filename) of a track quality file.

Returns
Reformatted date of the track quality file.

Return type
datetime.date | datetime.datetime

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> trk.fmt_dtf_file_date(tq_date='02-12-2019')
datetime.date(2019, 12, 2)
```

Track.get_dtf_pathname

Track.**get_dtf_pathname**(*tid*, *tq_date*)

Get the path to a track quality file of a given track ID and date.

Parameters

- **tid** (*str*) – Track ID.
- **tq_date** (*str | datetime.date*) – Date of a track quality file.

Returns
Pathname of the track quality file of the given `tid` and `tq_date`.

Return type
str

Examples:

```
>>> from src.preprocessor import Track
>>> import os
>>> trk = Track()
>>> dtf_file_path = trk.get_dtf_pathname(tid='1100', tq_date='02-12-2019')
>>> os.path.isfile(dtf_file_path)
True
```

Track.import_dtf

Track.**import_dtf**(*tid*, *tq_date*, *update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import data of a track quality file (of a given track ID and date) into the project database.

Parameters

- **tid** (*int | str*) – Track ID.
- **tq_date** (*str*) – Date of the track quality file.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **confirmation_required** (*bool*) – Whether asking for confirmation to proceed; defaults to True.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to True.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> trk.import_dtf(tid='2100', tq_date='30-03-2020')
To import "2100_30-03-2020.dtf" into the table "Track"."Track_quality_2100_30-
 ↪03-2020"?
 [No]|Yes: yes
Importing the data into  ... Done.
```

Track.import_gauging_changes

Track.**import_gauging_changes**(*trk_date*, *update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import the shapefile (calibrated) data of UK tracks into the project database.

Parameters

- **trk_date** (*str*) – Data date of track shapefiles.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **confirmation_required** (*bool*) – Whether to ask for confirmation to proceed; defaults to True.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to True.

- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> gauging_changes = trk.read_gauging_changes(trk_date='202004')
>>> type(gauging_changes)
dict
>>> list(gauging_changes.keys())
['Links_added', 'Links_deleted', 'Nodes_added', 'Nodes_deleted']
>>> trk.import_gauging_changes(trk_date='202004', if_exists='replace')
To import the Network Model gauging changes (39pt1 to 39pt2) into the schema
 ↪"Track"?
 [No]|Yes: yes
Importing ...
    "Links_added" ... Done.
    "Links_deleted" ... Done.
    "Nodes_added" ... Done.
    "Nodes_deleted" ... Done.
```

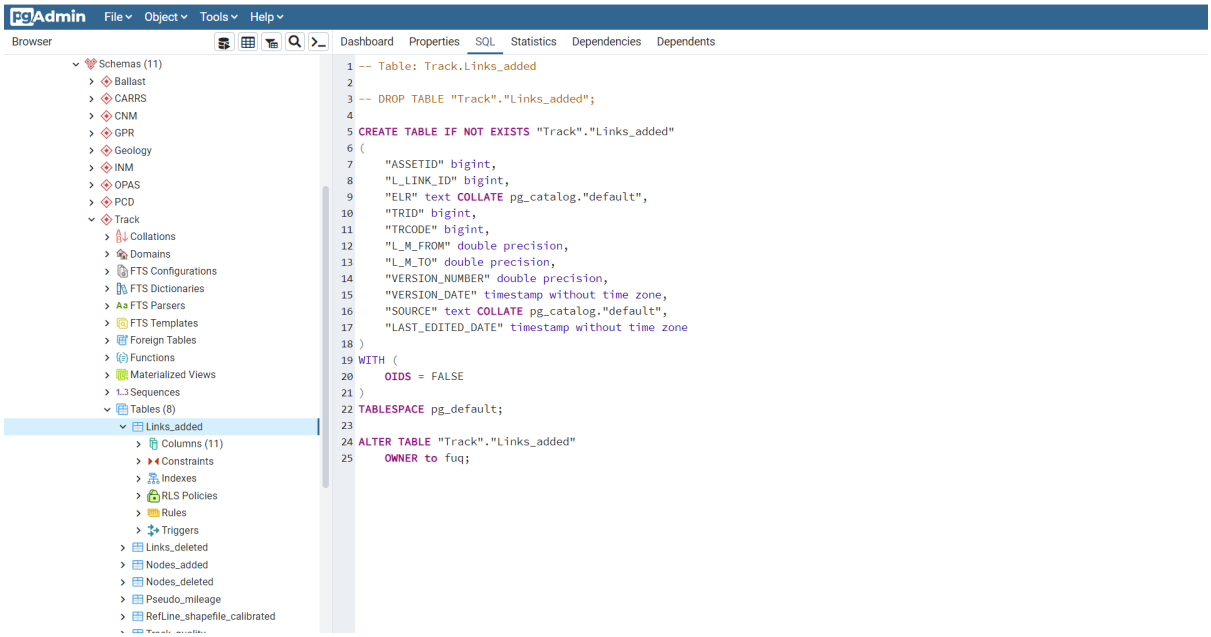


Figure 47: Snapshot of the "Track"."Links_added" table.

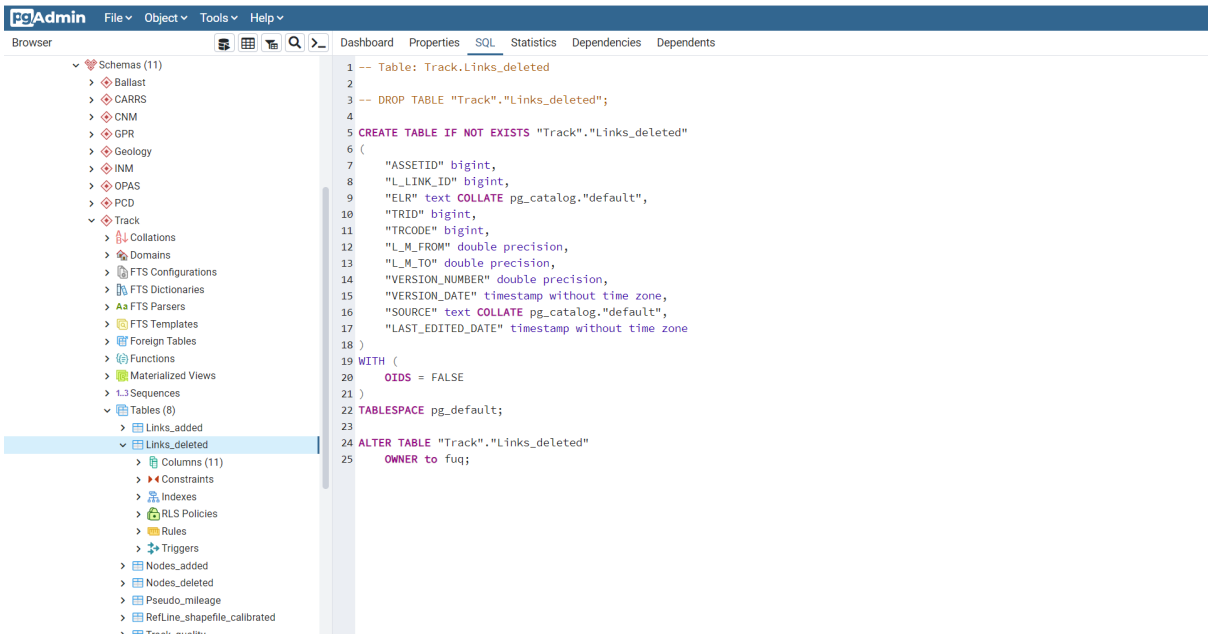


Figure 48: Snapshot of the "Track"."Links_deleted" table.

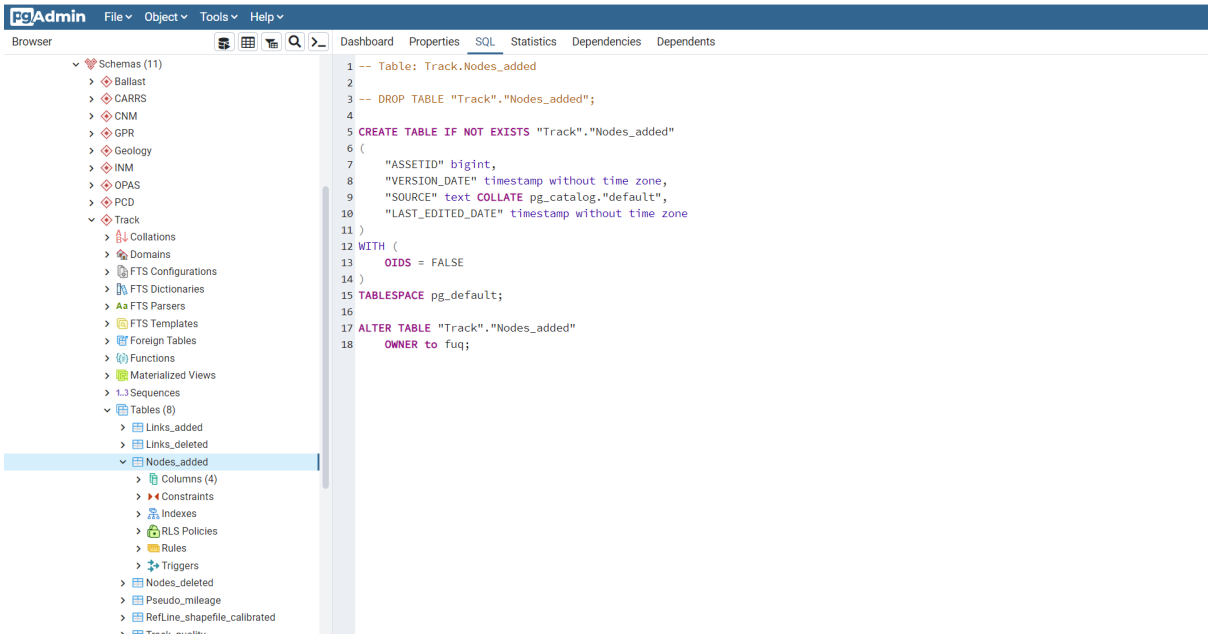


Figure 49: Snapshot of the "Track"."Nodes_added" table.

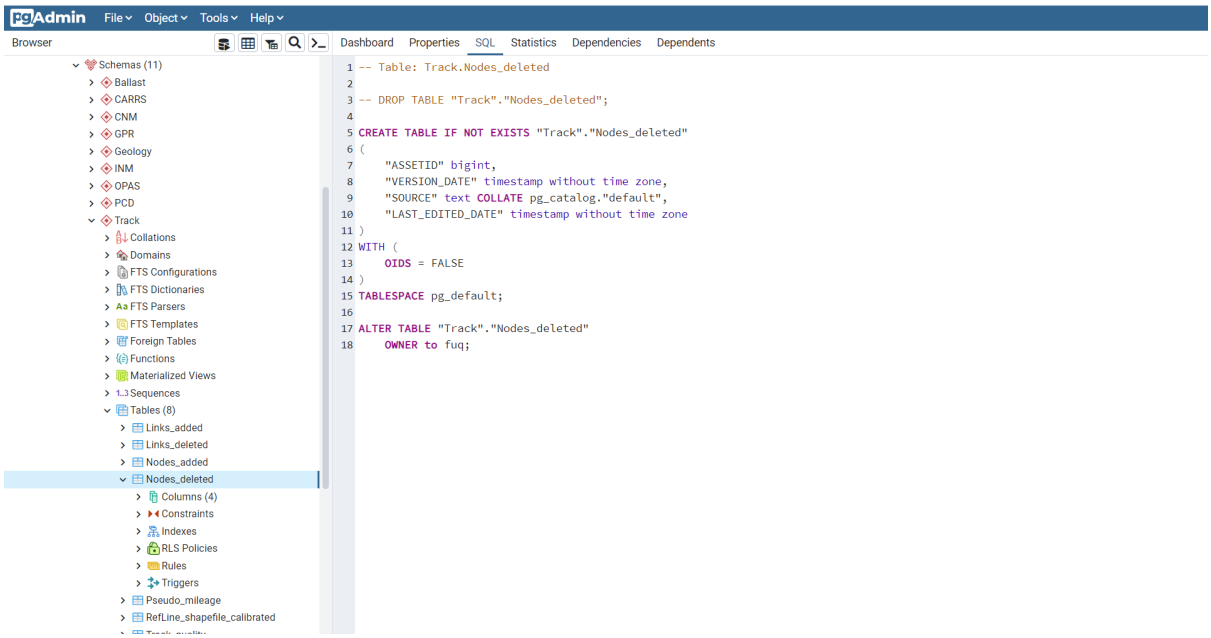


Figure 50: Snapshot of the "Track"."Nodes_deleted" table.

Track.import_pseudo_mileage_dict

`Track.`**`import_pseudo_mileage_dict`**(*elr=None*, *set_index=True*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import data of (pseudo) mileages into the project database.

Parameters

- **elr** (*str | list | None*) – (One) ELR; defaults to `None`.
- **set_index** (*bool*) – Whether to set `'pseudo_geometry'` to be the index; defaults to `True`.
- **confirmation_required** (*bool*) – Whether asking for confirmation to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to `True`.
- **kwargs** – [Optional] additional parameters for the pyhelpers.dbms.PostgreSQL.import_data method.

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> trk.import_pseudo_mileage_dict(elr=['ECM7', 'ECM8'])
To import pseudo mileages (ECM7, ECM8) into the schema "Track"?
 [No]|Yes: yes
Importing the data ... Done.
```

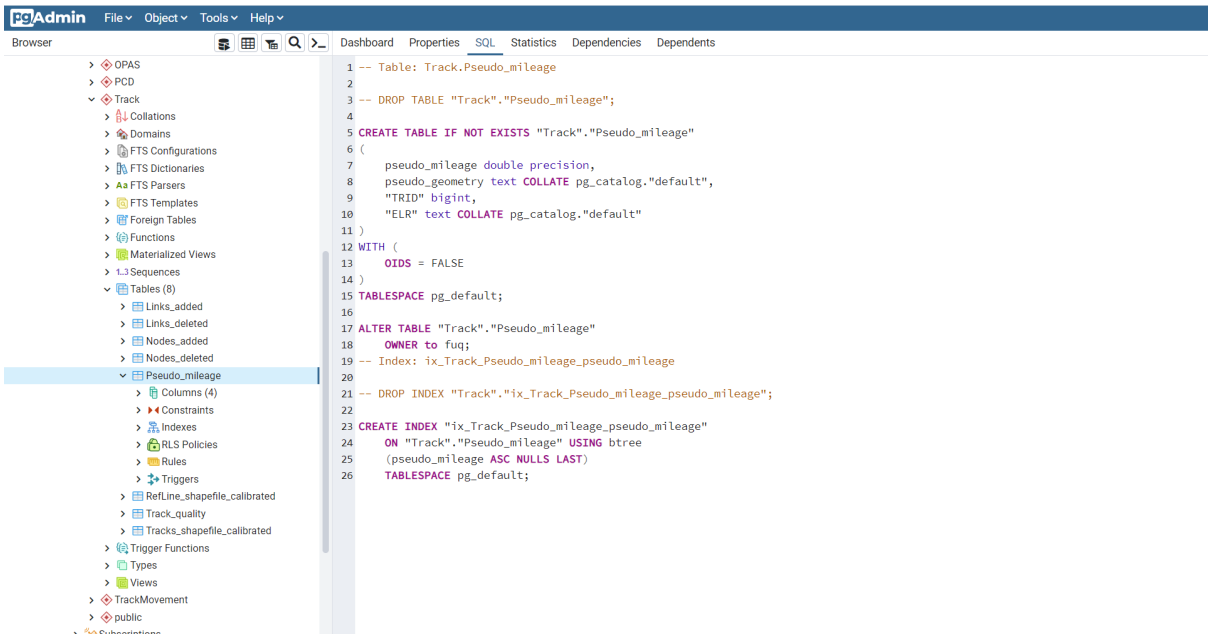


Figure 51: Snapshot of the "Track"."Pseudo_mileage" table.

Track.import_ref_line_shp

Track.**import_ref_line_shp**(*trk_date*, *update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import the shapefile (calibrated) of the reference line into the project database.

Parameters

- **trk_date** (*str*) – Data date of track shapefiles.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **confirmation_required** (*bool*) – Whether to ask for confirmation to proceed; defaults to True.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to True.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> trk.import_ref_line_shp(trk_date='202004')
To import ref line data into the table "Track"."RefLine_shapefile_calibrated"?
 [No]|Yes: yes
Importing the data ... Done.
```

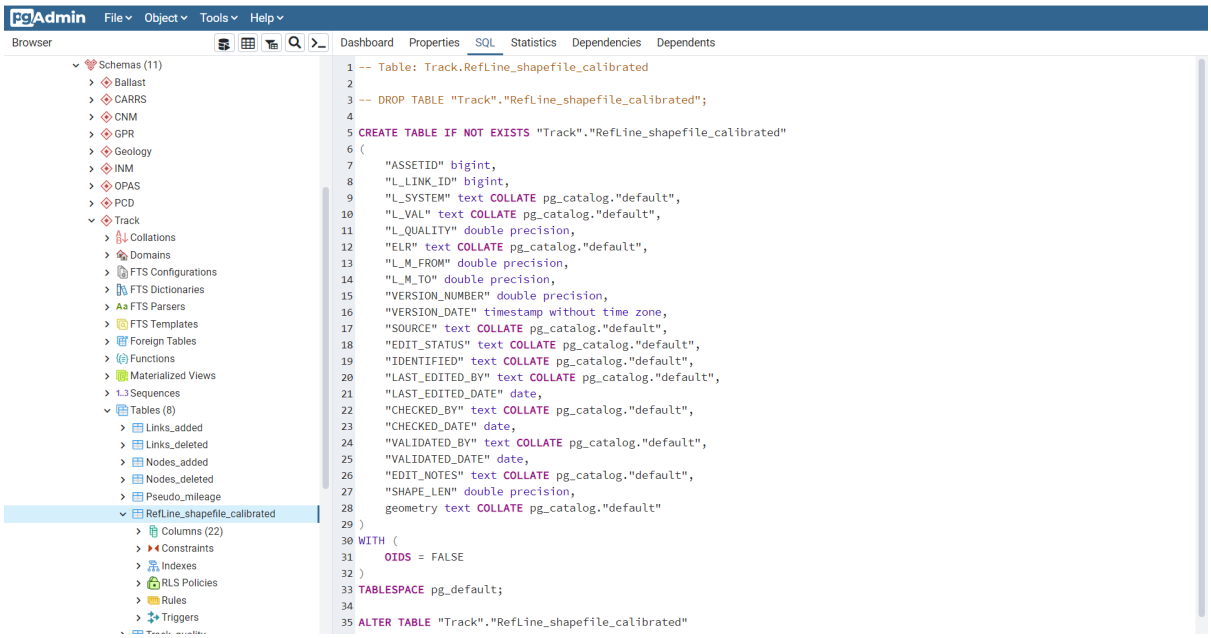


Figure 52: Snapshot of the "Track"."RefLine_shapefile_calibrated" table.

Track.import_track_quality

Track.**import_track_quality**(*update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import data of all track quality files into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **confirmation_required** (*bool*) – Whether asking for confirmation to proceed; defaults to True.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to True.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> trk.import_track_quality()
To import track quality files into the table "Track"."Track_quality"?
 [No]|Yes: yes
Importing the data ... Done.
```

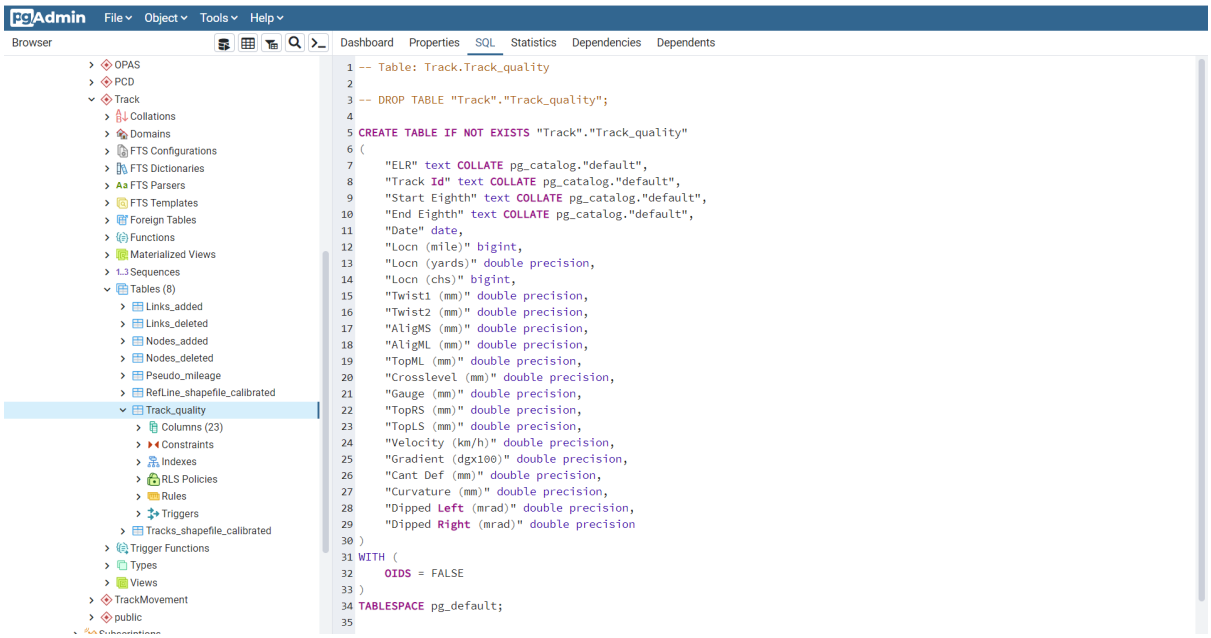


Figure 53: Snapshot of the "Track"."Track_quality" table.

Track.import_tracks_shapefiles

Track.**import_tracks_shapefiles**(*trk_date*, *update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import data of track shapefiles into the project database.

Parameters

- **trk_date** (*str*) – Data date of track shapefiles.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **confirmation_required** (*bool*) – Whether to ask for confirmation to proceed; defaults to True.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to True.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> trk.import_tracks_shapefiles(trk_date='202004', if_exists='replace')
To import data of track shapefiles into the schema "Track"?
 [No]|Yes: yes
Importing ...
    Shapefile (calibrated) of tracks of the whole UK ... Done.
    Network Model gauging changes (39pt1 to 39pt2) ... Done.
    Shapefile (calibrated) of reference line ... Done.
```

See also:

- Examples for the mothds `import_gauging_changes()`, `import_ref_line_shp()` and `import_tracks_shp()`.

Track.import_tracks_shp

`Track.`**`import_tracks_shp`**(*trk_date*, *update=False*, *confirmation_required=True*, *verbose=True*, ***kwargs*)

Import the shapefile (calibrated) data of UK tracks into the project database.

Parameters

- **`trk_date`** (*`str`*) – Data date of track shapefiles.
- **`update`** (*`bool`*) – Whether to reprocess the original data file(s); defaults to `False`.
- **`confirmation_required`** (*`bool`*) – Whether to ask for confirmation to proceed; defaults to `True`.
- **`verbose`** (*`bool | int`*) – Whether to print relevant information in the console; defaults to `True`.
- **`kwargs`** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> trk.import_tracks_shp(trk_date='202004')
To import track shapefile into the table "Track"."Tracks_shapefile_calibrated"?
 [No]|Yes: yes
Importing the data ... Done.
```

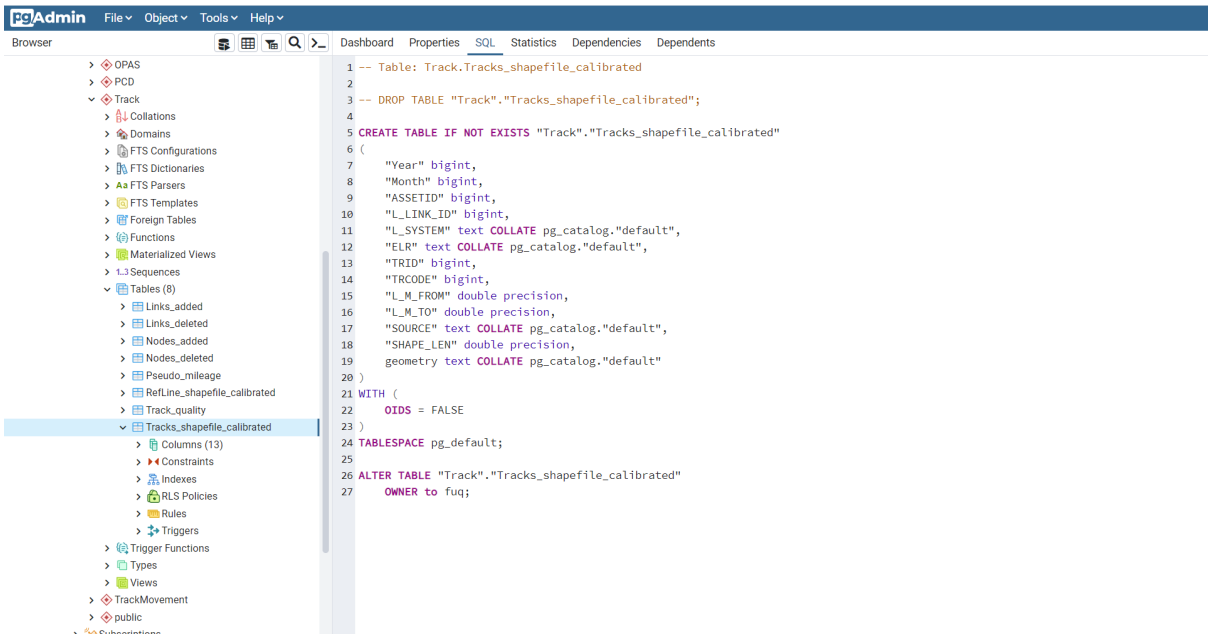


Figure 54: Snapshot of the "Track"."Tracks_shapefile_calibrated" table.

Track.load_pseudo_mileage_dict

`Track.`**`load_pseudo_mileage_dict`**(*elr=None*, ***kwargs*)

Load the data of (pseudo) mileages from the project database.

Parameters

- **elr** (*str | list | tuple | None*) – Engineer's Line Reference(s); defaults to `None`.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.read_sql_query.

Returns

Data of pseudo mileages (based on track shapefile).

Return type

dict

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> pseudo_track_mileage_dict = trk.load_pseudo_mileage_dict(elr=['ECM7', 'ECM8
↪'])
>>> type(pseudo_track_mileage_dict)
dict
>>> list(pseudo_track_mileage_dict.keys())
['ECM7', 'ECM8']
>>> list(pseudo_track_mileage_dict['ECM8'].keys())[:5]
[1100, 1200, 1700, 1900, 1901]
>>> pseudo_track_mileage_dict['ECM8'][1100].shape
(94899, 1)
```

Track.load_track_quality

Track.**load_track_quality**(*elr=None*, *tq_date=None*, ***kwargs*)

Load the data of track quality files from the project database.

Parameters

- **elr** (*str | list | tuple | None*) – Engineer's Line Reference(s); defaults to None.
- **tq_date** (*str | list | None*) – Date of a track quality file, formatted as 'YYYY-MM-DD'; defaults to None.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.read_sql_query.

Returns

Data of track quality files.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> tqf_data = trk.load_track_quality(elr=['ECM7', 'ECM8'])
>>> tqf_data.shape
(1375430, 23)
```

Track.load_tracks_shp

Track.**load_tracks_shp**(*trk_date=None*, *elr=None*, ***kwargs*)

Load the shapefile (calibrated) data of UK tracks from the project database.

Parameters

- **trk_date** (*str | int | None*) – Data date of track shapefiles.
- **elr** (*str | list | tuple | None*) – Engineer's Line Reference(s); defaults to None.
- **kwargs** – [Optional] additional parameters for the method pyhelpers.dbms.PostgreSQL.read_sql_query.

Returns

Shapefile (calibrated) data of UK tracks.

Return type

pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> trk_shp = trk.load_tracks_shp(trk_date='202004', elr=['ECM7', 'ECM8'])
>>> trk_shp.shape
(440, 13)
```

Track.make_pseudo_mileage_dict

Track.**make_pseudo_mileage_dict**(*elr=None*, *set_index=True*)

Make a dictionary of (pseudo) mileages for all available track IDs based on track shapefiles.

Parameters

- **elr** (*str | list | None*) – (One) ELR; defaults to None.
- **set_index** (*bool*) – Whether to set 'pseudo_geometry' to be the index for the dataframe of pseudo track mileage; defaults to True.

Returns

Mileages for all available track IDs from track shapefiles.

Return type

dict

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> pseudo_track_mileage_dict = trk.make_pseudo_mileage_dict(elr='ECM8')
>>> list(pseudo_track_mileage_dict.keys())
['ECM8']
>>> list(pseudo_track_mileage_dict['ECM8'].keys())[:5]
[1100, 1200, 1700, 1900, 1901]
>>> pseudo_track_mileage_dict['ECM8'][1100].shape
(94899, 1)
```

Track.parse_dtf

Track.**parse_dtf**(*path_to_dtf*)

Parse a track quality file (.dtf format).

Parameters

path_to_dtf (*str*) – Path to the track quality .dtf file.

Returns

Data parsed from the track quality .dtf file.

Return type

dict

Examples:

```
>>> from src.preprocessor import Track
>>> import os
>>> trk = Track()
>>> path_to_dtf_file = trk.dtf_pathnames[0]
>>> os.path.isfile(path_to_dtf_file)
True
>>> dtf_dat = trk.parse_dtf(path_to_dtf_file)
>>> type(dtf_dat)
dict
>>> list(dtf_dat.keys())
['TQ', 'ELR', 'Track Id', 'Start Eighth', 'End Eighth', 'Date(DD/MM/YYYY)']
>>> dtf_dat['TQ'].shape
(334844, 18)
```

Track.read_dtf

Track.**read_dtf**(*tid*, *tq_date*, *update=False*, *verbose=False*)

Read track quality files (.dtf) for a given pair of track ID and date from a local directory.

Parameters

- **tid** (*int | str*) – Track ID.
- **tq_date** (*str*) – Date of the track quality file.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to False.

Returns

Data of the track quality file, or None if the file is not found.

Return type

dict | None

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> trk.track_quality_tid
['1100', '2100']
>>> trk.dtf_data_dates
{'1100': ['14-10-2019', '02-12-2019', '27-04-2020'],
 '2100': ['14-10-2019', '06-01-2020', '30-03-2020']}
>>> # tqf_data = trk.read_dtf('1100', '14-10-2019', update=True, verbose=True)
>>> tqf_data = trk.read_dtf(tid='1100', tq_date='14-10-2019')
>>> type(tqf_data)
dict
>>> list(tqf_data.keys())
['TQ', 'ELR', 'Track Id', 'Start Eighth', 'End Eighth', 'Date(DD/MM/YYYY)']
>>> tqf_data['TQ'].shape
(334844, 18)
```

Track.read_gauging_changes

Track.**read_gauging_changes**(*trk_date*, *update=False*, *verbose=False*)

Read data of network model changes on '39pt1_to_39pt2_gauging' from a local directory.

Parameters

- **trk_date** (*str*) – Data date of track shapefiles.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to False.

Returns

Data of the network model changes on '39pt1_to_39pt2_gauging'.

Return type

dict

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> trk_date = '202004'
>>> gauging_changes = trk.read_gauging_changes(trk_date)
>>> # gauging_changes = trk.read_gauging_changes(trk_date, update=True,␣
↪verbose=True)
>>> type(gauging_changes)
dict
>>> list(gauging_changes.keys())
['Links_added', 'Links_deleted', 'Nodes_added', 'Nodes_deleted']
>>> gauging_changes['Links_added'].shape
(3, 11)
>>> gauging_changes['Links_deleted'].shape
(17, 11)
>>> gauging_changes['Nodes_added'].shape
(2, 4)
>>> gauging_changes['Nodes_deleted'].shape
(13, 4)
```

Track.read_ref_line_shp

Track.**read_ref_line_shp**(*trk_date*, *update=False*, *verbose=False*, ***kwargs*)

Read the shapefile (calibrated) of the reference line from a local directory.

Parameters

- **trk_date** (*str*) – Data date of track shapefiles.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.

- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to `False`.
- **kwargs** – [Optional] additional parameters for the method pydriosm.reader.SHPReadParse.read_shp.

Returns
Tabular data of the shapefile (calibrated) of the reference line.

Return type
geopandas.GeoDataFrame | pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> ref_line = trk.read_ref_line_shp(trk_date='202004')
>>> # ref_line = trk.read_ref_line_shp(trk_date='202004', update=True,␣
↪verbose=True)
>>> type(ref_line)
geopandas.geodataframe.GeoDataFrame
>>> ref_line.shape
(1583, 22)
```

Track.read_track_quality

Track.**read_track_quality**(*update=False*, *verbose=False*)

Read data of all available track quality files from a local directory.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to `False`.

Returns
Data of all available track quality files.

Return type
pandas.DataFrame

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> # tqf_data = trk.read_track_quality(update=True, verbose=True)
>>> tqf_data = trk.read_track_quality()
>>> tqf_data.shape
(2093003, 23)
```

Track.read_tracks_shp

Track.**read_tracks_shp**(*trk_date*, *update=False*, *verbose=False*, ***kwargs*)

Read the shapefile (calibrated) data of tracks of the whole UK from a local directory.

Parameters

- **trk_date** (*str*) – Data date of track shapefiles.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information in the console; defaults to False.
- **kwargs** – [Optional] additional parameters for the method pydriosm.reader.SHPReadParse.read_shp.

Returns

Tabular data of the shapefile (calibrated) of tracks of the entire UK.

Return type

geopandas.GeoDataFrame | pandas.DataFrame | None

Examples:

```
>>> from src.preprocessor import Track
>>> trk = Track()
>>> # trk_shp = trk.read_tracks_shp(trk_date='202004', update=True,␣
↪verbose=True)
>>> trk_shp = trk.read_tracks_shp(trk_date='202004')
>>> type(trk_shp)
geopandas.geodataframe.GeoDataFrame
>>> trk_shp.shape
(49667, 11)
>>> # example_record = trk_shp.loc[[10545], :]
>>> # example_record
```

2.3 shaft

Further processing and exploration of the data that are preprocessed by the subpackage *preprocessor* to generate a comprehensive data set for the development of a machine learning model on track fixity.

The data processing includes three tasks:

- Calculation of track movement.
- Collation of the data of variables that influence the track movement.
- Integration of data sets of the calculated track movement and the influencing variables.

2.3.1 Classes

PCDHandler([db_instance])	Explore and process the point cloud data.
KRDZGear([elr, db_instance])	Further process the KRDZ data (within the point cloud category).
TrackMovement([elr, db_instance])	Calculate track movement - the target to be predicted.
FeatureCollator([elr, db_instance, verbose])	Collate data on various features for developing a machine learning model to predict track movement (i.e. track fixity parameters).

PCDHandler

class src.shaft.**PCDHandler**(*db_instance=None*)

Explore and process the point cloud data.

Parameters
db_instance (*src.utils.TrackFixityDB | None*) – PostgreSQL database instance; defaults to None.

Examples:

```
>>> from src.shaft import PCDHandler
>>> pcdh = PCDHandler()
>>> pcdh.NAME
'Handler of preprocessed point cloud data'
```

Attributes:

NAME	Descriptive name of the class.

PCDHandler.NAME

PCDHandler.**NAME: str = 'Handler of preprocessed point cloud data'**

Descriptive name of the class.

Methods:

get_pcd_tile_mpl_path(tile_xy[, pcd_date])	Make a matplotlib.path.Path object for a tile for the point cloud data.
get_pcd_tile_polygon(tile_xy[, pcd_date, ...])	Make a polygon for a given (X, Y) in reference to a tile.
o3d_transform(xyz_rgb_data[, greyscale, ...])	Transform the numpy array of point cloud (XYZ and RGB) data to Open3D object(s) (open3d.geometry.PointCloud).
view_pcd_dgn_shp_polyline(tile_xy, pcd_date)	Visualise polyline data of the DGN-converted shapefile of point cloud data.
view_pcd_example(tile_xy[, pcd_dates, ...])	Visualise a sample (tile) of point cloud data.

PCDHandler.get_pcd_tile_mpl_path

PCDHandler.**get_pcd_tile_mpl_path**(*tile_xy*, *pcd_date=None*, ***kwargs*)

Make a matplotlib.path.Path object for a tile for the point cloud data.

Parameters

- **tile_xy** (*tuple | list | str*) – X and Y coordinates in reference to a tile for the point cloud data.
- **pcd_date** (*str | int | None*) – Date of the point cloud data; defaults to None.
- **kwargs** – [Optional] parameter used by matplotlib.path.Path.

Returns

matplotlib.path.Path object for the tile labelled tile_xy.

Return type

matplotlib.path.Path

Examples:

```
>>> from src.shaft import PCDHandler
>>> pcdh = PCDHandler()
>>> # Make a polygon (i.e. a matplotlib.Path instance) of an example tile
>>> tile_frame = pcdh.get_pcd_tile_mpl_path(tile_xy=(340100, 674000), pcd_date=
↪'202004')
>>> type(tile_frame)
matplotlib.path.Path
>>> tile_frame
Path(array([[340100.0000, 674000.0000],
       [340100.0000, 674100.0000],
       [340200.0000, 674100.0000],
       [340200.0000, 674000.0000],
       [340100.0000, 674000.0000]]), array([ 1,  2,  2,  2, 79], dtype=uint8))
```

PCDHandler.get_pcd_tile_polygon

PCDHandler.**get_pcd_tile_polygon**(*tile_xy*, *pcd_date=None*, *as_geom=True*)

Make a polygon for a given (X, Y) in reference to a tile.

Parameters

- **tile_xy** (*tuple | list | str*) – X and Y coordinates in reference to a tile for the point cloud data.
- **pcd_date** (*str | int | None*) – Date of the point cloud data; defaults to None.
- **as_geom** (*bool*) – Whether to return the polygon as a geometry object; defaults to True.

Returns

Polygon for the given (X, Y) in reference to a tile.

Return type

shapely.geometry.Polygon | numpy.ndarray

Examples:

```
>>> from src.shaft import PCDHandler
>>> pcdh = PCDHandler()
>>> tile_x_y = (340500, 674200)
>>> tile_polygon = pcdh.get_pcd_tile_polygon(tile_xy=tile_x_y)
>>> type(tile_polygon)
shapely.geometry.polygon.Polygon
>>> print(tile_polygon.wkt)
POLYGON ((340500 674200, 340500 674300, 340600 674300, 340600 674200, 340500␣
↪674200))
>>> tile_polygon = pcdh.get_pcd_tile_polygon(tile_xy=tile_x_y, as_geom=False)
>>> type(tile_polygon)
numpy.ndarray
>>> tile_polygon
array([[340500, 674200],
       [340500, 674300],
       [340600, 674300],
       [340600, 674200],
       [340500, 674200]])
```

Illustration:

```
import matplotlib.pyplot as plt
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg', font_name='Times New Roman')

fig = plt.figure(constrained_layout=True)
ax = fig.add_subplot()
ax.xaxis.set_ticks([tile_x_y[0], tile_x_y[0] + 100])
ax.yaxis.set_ticks([tile_x_y[1], tile_x_y[1] + 100])

c1, c2 = plt.get_cmap('tab10').colors[:2]
```

(continues on next page)

(continued from previous page)

```
ax.plot(tile_polygon[:, 0], tile_polygon[:, 1], linewidth=3)
tile_label = f'Tile for {tile_x_y}'
ax.scatter([], [], marker='s', s=200, fc='none', ec=c1, lw=3, label=tile_label)
ax.scatter(tile_x_y[0], tile_x_y[1], s=100, color=c2, label=f'{tile_x_y}')

ax.legend(loc='best')

fig.show()

# from pyhelpers.store import save_figure
# fig_pathname = "docs/source/_images/pcdh_get_pcd_tile_polygon_demo"
# save_figure(fig, f"{fig_pathname}.svg", verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", verbose=True)
```

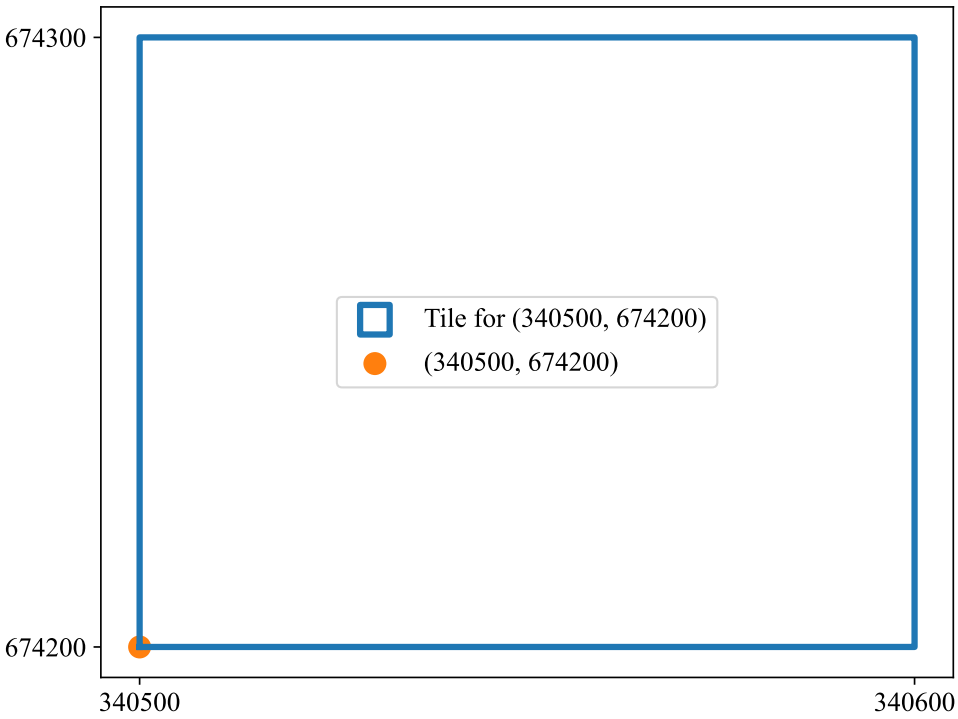


Figure 55: Tile of (340100, 674000).

PCDHandler.o3d_transform

static PCDHandler.**o3d_transform**(*xyz_rgb_data*, *greyscale=False*, *voxel_size=None*)

Transform the numpy array of point cloud (XYZ and RGB) data to Open3D object(s) (open3d.geometry.PointCloud).

Parameters

- **xyz_rgb_data** (*dict*) – XYZ and RGB data.
- **voxel_size** (*float | None*) – Voxel size for downsampling; defaults to None.

- **greyscale** (*bool*) – Whether to transform the colour data to greyscale; defaults to `False`.

Returns
Open3D objects of point cloud data.

Return type
tuple

Examples:

```
>>> from src.shaft import PCDHandler
>>> from pyhelpers.settings import np_preferences
>>> np_preferences()
>>> pcdh = PCDHandler()
>>> tile_xy = (340500, 674200)
>>> rows_limit = 1000
>>> # Data of '202004'
>>> xyz_rgb_dat = pcdh.load_laz(tile_xy, pcd_dates=['202004'], limit=rows_limit)
>>> type(xyz_rgb_dat)
dict
>>> list(xyz_rgb_dat.keys())
['202004']
>>> len(xyz_rgb_dat['202004'])
2
>>> type(xyz_rgb_dat['202004'][0])
numpy.ndarray
>>> xyz_rgb_dat['202004'][0].shape
(1000, 3)
>>> type(xyz_rgb_dat['202004'][1])
numpy.ndarray
>>> xyz_rgb_dat['202004'][1].shape
(1000, 3)
>>> # Original data of all dates available
>>> xyz_rgb_dat = pcdh.load_laz(tile_xy, limit=rows_limit)
>>> list(xyz_rgb_dat.keys())
['201910', '202004']
>>> # Transform the array data to open3d.geometry.PointCloud class
>>> o3d_pcd_lst = pcdh.o3d_transform(xyz_rgb_dat)
>>> o3d_pcd_lst
[PointCloud with 1000 points., PointCloud with 1000 points.]
>>> # Downsample the data with voxels
>>> o3d_pcd_ds_lst = pcdh.o3d_transform(xyz_rgb_dat, voxel_size=0.05)
>>> o3d_pcd_ds_lst
[PointCloud with 915 points., PointCloud with 122 points.]
```

PCDHandler.view_pcd_dgn_shp_polyline

PCDHandler.**view_pcd_dgn_shp_polyline**(*tile_xy*, *pcd_date*, *projection='3d'*, *add_title=False*, *save_as=None*, *dpi=600*, *verbose=False*, ***kwargs*)

Visualise polyline data of the DGN-converted shapefile of point cloud data.

Parameters

- **tile_xy** (*tuple | list | str*) – X and Y coordinates in reference to a tile for the point cloud data.
- **pcd_date** (*str | int*) – Date of the point cloud data.
- **projection** (*str | None*) – Projection type of the subplot; defaults to `'3d'`.
- **add_title** (*bool*) – Whether to add a title to the plot; defaults to `False`.
- **save_as** (*str | None*) – File format that the view is saved as; defaults to `None`.
- **dpi** (*int | None*) – DPI for saving image; defaults to `600`.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to `False`.
- **kwargs** – [Optional] additional parameters for the function matplotlib.pyplot.scatter.

Examples:

```
>>> from src.shaft import PCDHandler
>>> from pyhelpers.settings import mpl_preferences
>>> mpl_preferences(backend='TkAgg')
>>> pcdh = PCDHandler()
>>> tile_xy = (340500, 674200)
>>> pcd_date = '202004'
>>> # pcdh.view_pcd_dgn_shp_polyline(
... #     tile_xy=tile_xy, pcd_date=pcd_date, projection='3d', s=5, save_as=".
 ↪svg",
... #     verbose=True)
>>> pcdh.view_pcd_dgn_shp_polyline(tile_xy, pcd_date, projection='3d', s=5)
```

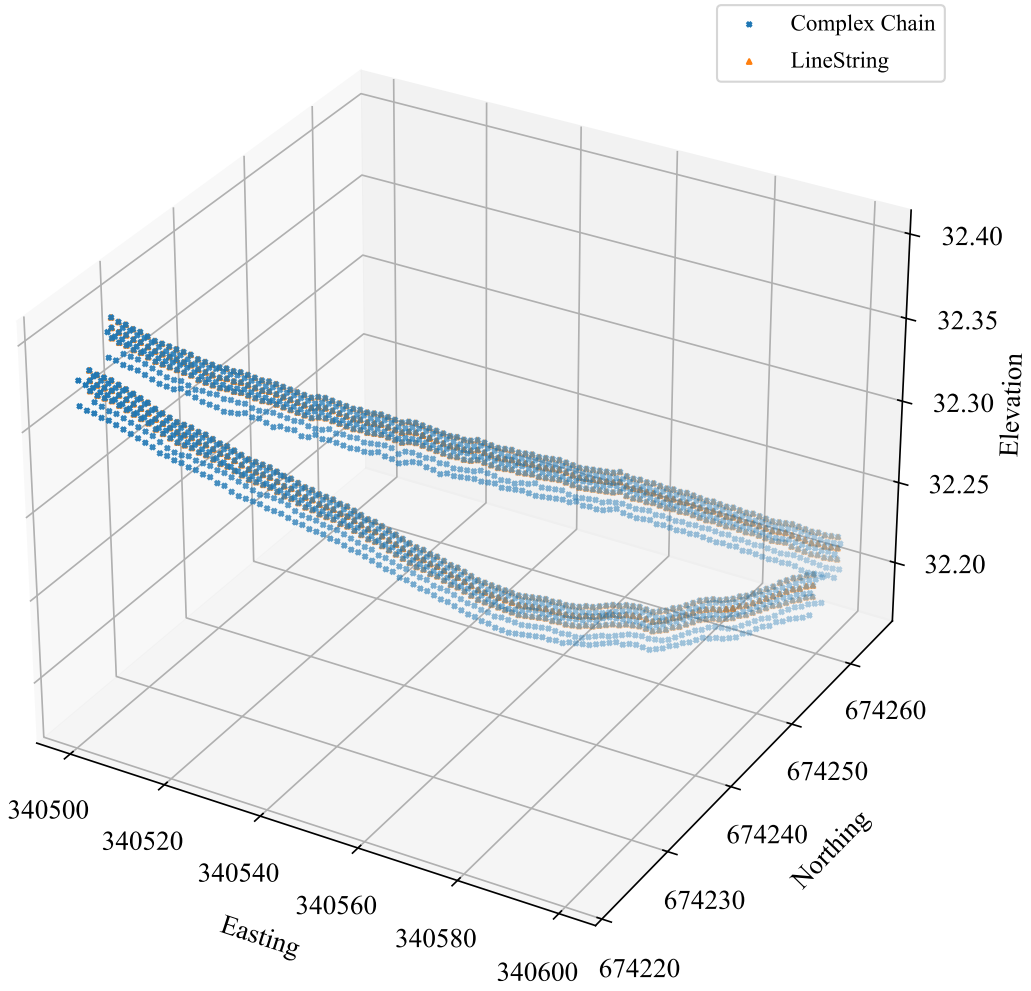


Figure 56: DGN-PointCloud data (3D) of Tile (340500, 674200) in April 2020.

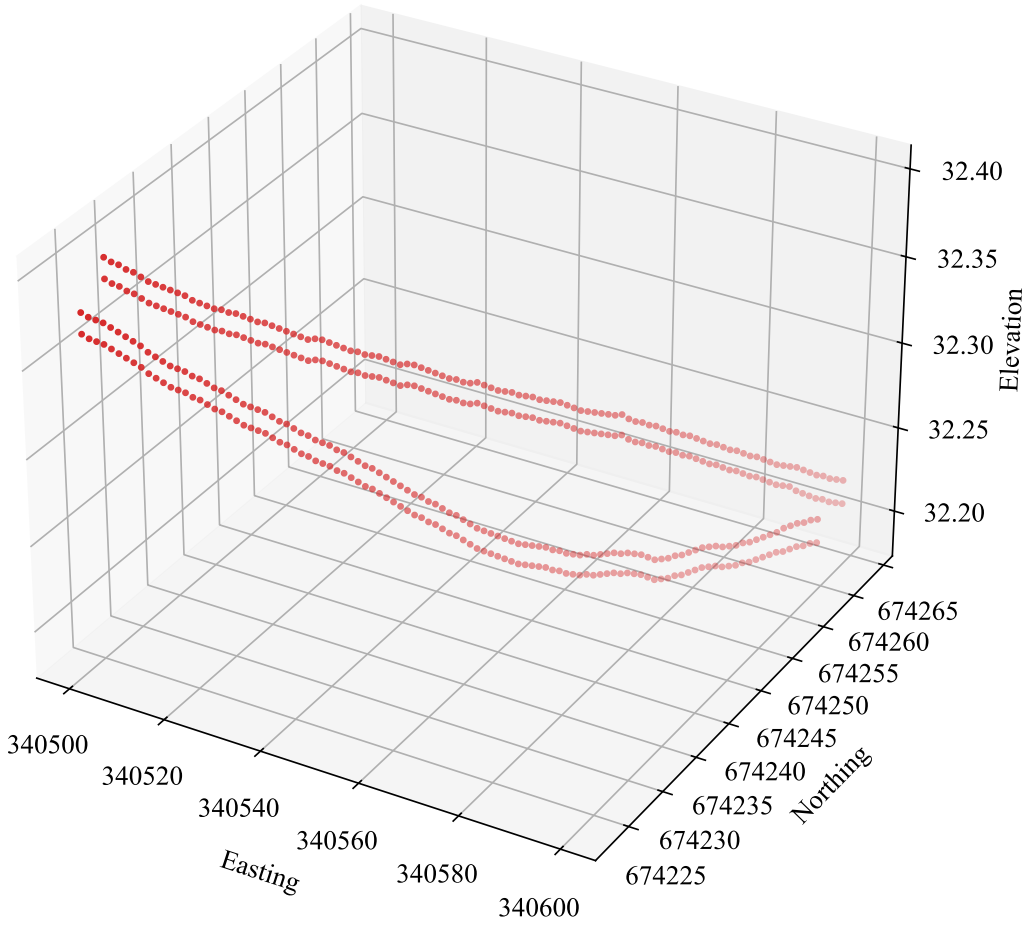


Figure 57: Common points (3D) between 'Complex Chain' and 'LineString' entities of polyline of Tile (340500, 674200) in April 2020.

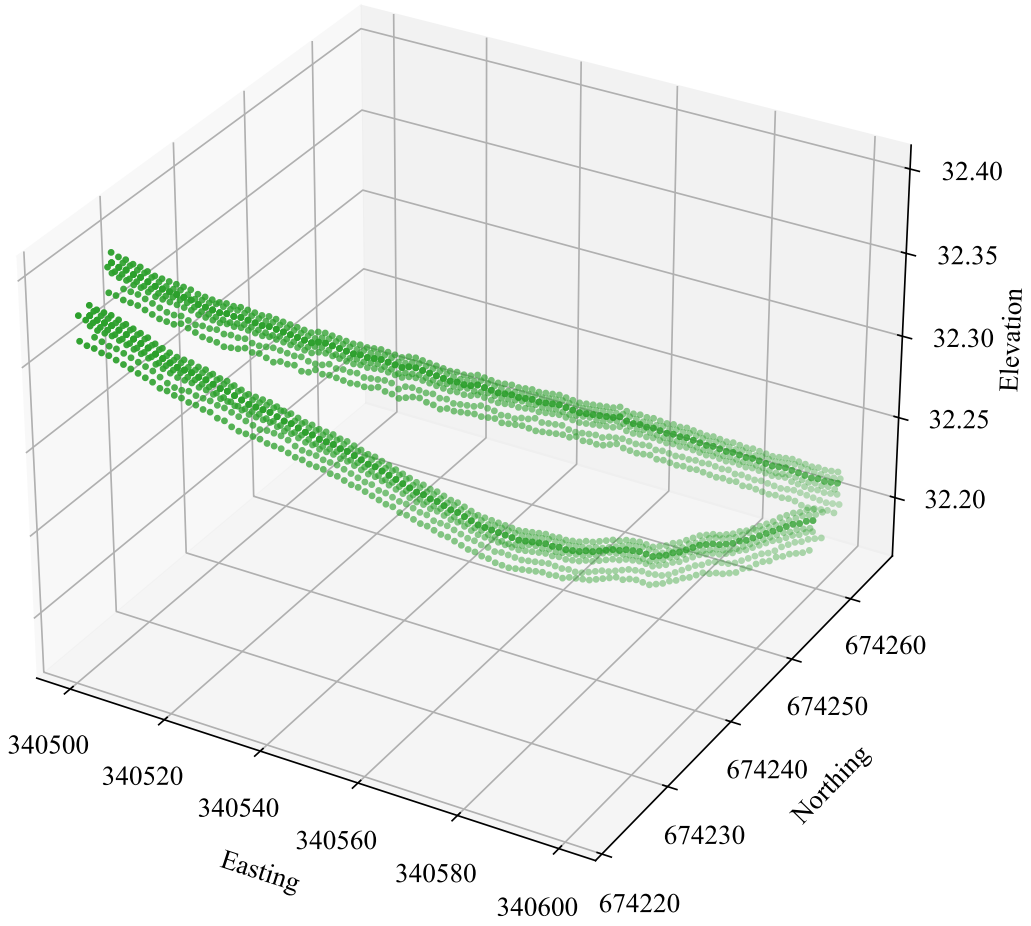


Figure 58: Unique points (3D) between 'Complex Chain' and 'LineString' entities of polyline of Tile (340500, 674200) in April 2020.

```
>>> # 2D plots - Vertical view
>>> # pcdh.view_pcd_dgn_shp_polyline(
... #     tile_xy, pcd_date, projection=None, s=5, save_as=".svg", verbose=True)
>>> pcdh.view_pcd_dgn_shp_polyline(tile_xy, pcd_date, projection=None, s=5)
```

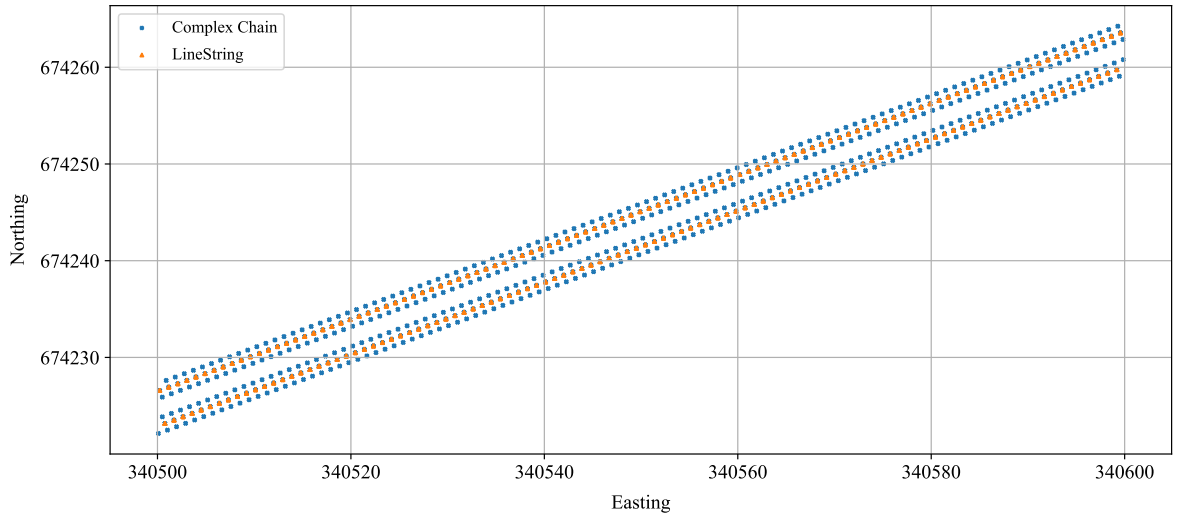


Figure 59: DGN-PointCloud data of Tile (340500, 674200) in April 2020.

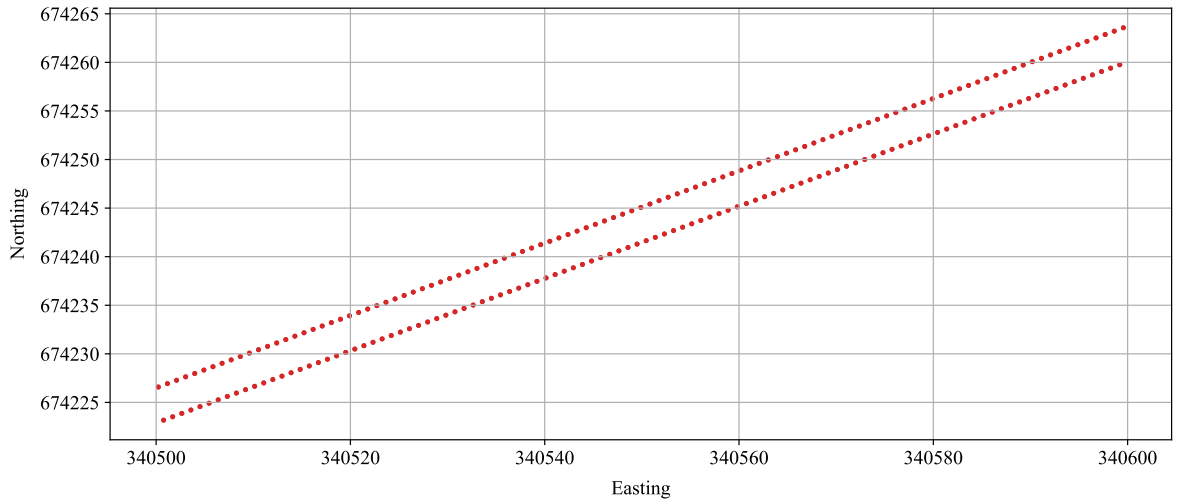


Figure 60: Common points between 'Complex Chain' and 'LineString' entities of polyline of Tile (340500, 674200) in April 2020.

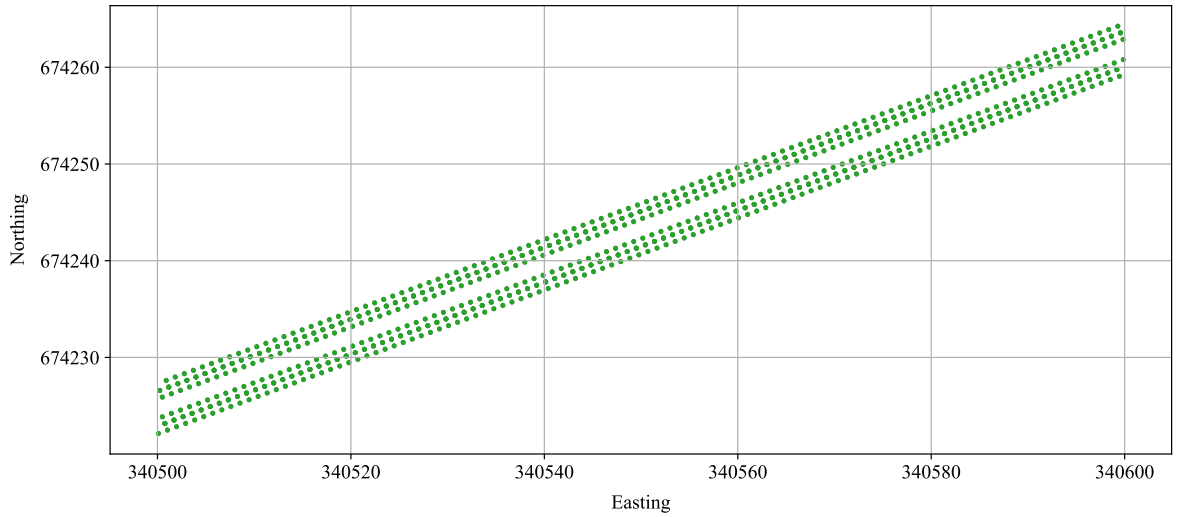


Figure 61: Unique points between 'Complex Chain' and 'LineString' entities of polyline of Tile (340500, 674200) in April 2020.

PCDHandler.view_pcd_example

PCDHandler.**view_pcd_example**(*tile_xy*, *pcd_dates=None*, *limit=None*, *voxel_size=None*, *greyscale=False*, *gs_coef=1.2*, *save_as=None*, *verbose=False*, *ret_sample=True*, ***kwargs*)

Visualise a sample (tile) of point cloud data.

Parameters

- **tile_xy** (*tuple | list | str*) – X and Y coordinates in reference to a tile for the point cloud data.
- **pcd_dates** (*str | list | None*) – Date(s) of the point cloud data; defaults to `None`.
- **voxel_size** (*float | None*) – Voxel size for downsampling; defaults to `0.05`.
- **greyscale** (*bool*) – Whether to transform the colour data to greyscale; defaults to `False`.
- **gs_coef** (*float*) – Coefficient associated with intensity (only if `greyscale=True`), defaults to `1.2`.
- **limit** (*int*) – Limit on the number of rows to query from the database; defaults to `None`.
- **save_as** (*str | None*) – File format that the view is saved as; defaults to `None`.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to `False`.

- **ret_sample** (*bool*) – Whether to return the sample data.
- **kwargs** – [Optional] parameters for open3d.visualization.draw_geometries.

Returns
The Open3D object(s) of the sample point cloud data (only if `ret_sample=True`).

Return type
tuple

Examples:

```
>>> from src.shaft import PCDHandler
>>> pcdh = PCDHandler()
>>> xy_tile = (340500, 674200)
>>> pcd1 = pcdh.view_pcd_example(xy_tile, '201910', greyscale=True, width=1000)
```

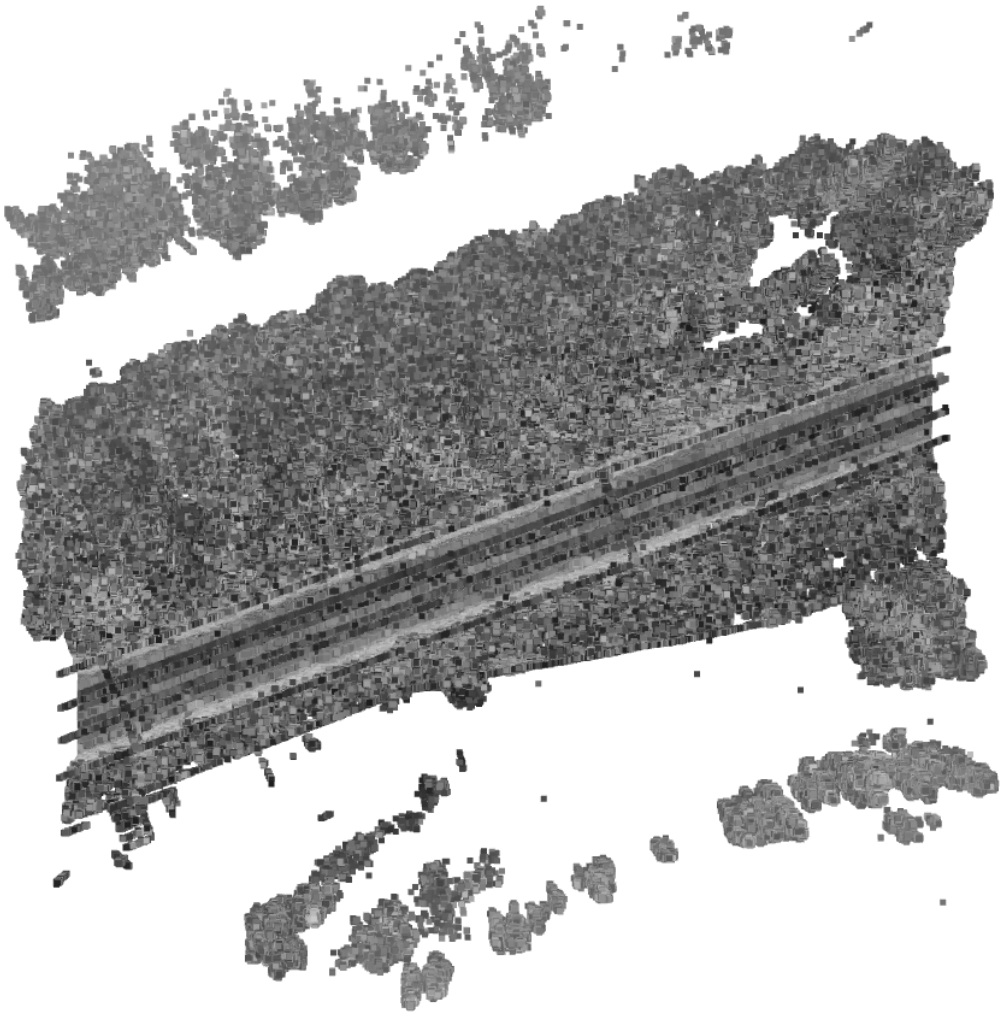


Figure 62: Point cloud data (greyscale, October 2019) of tile (340500, 674200).

```
>>> pcd1
[PointCloud with 54007138 points.]
```

```
>>> pcd2 = pcdh.view_pcd_example(xy_tile, pcd_dates='201910', width=1000)
```

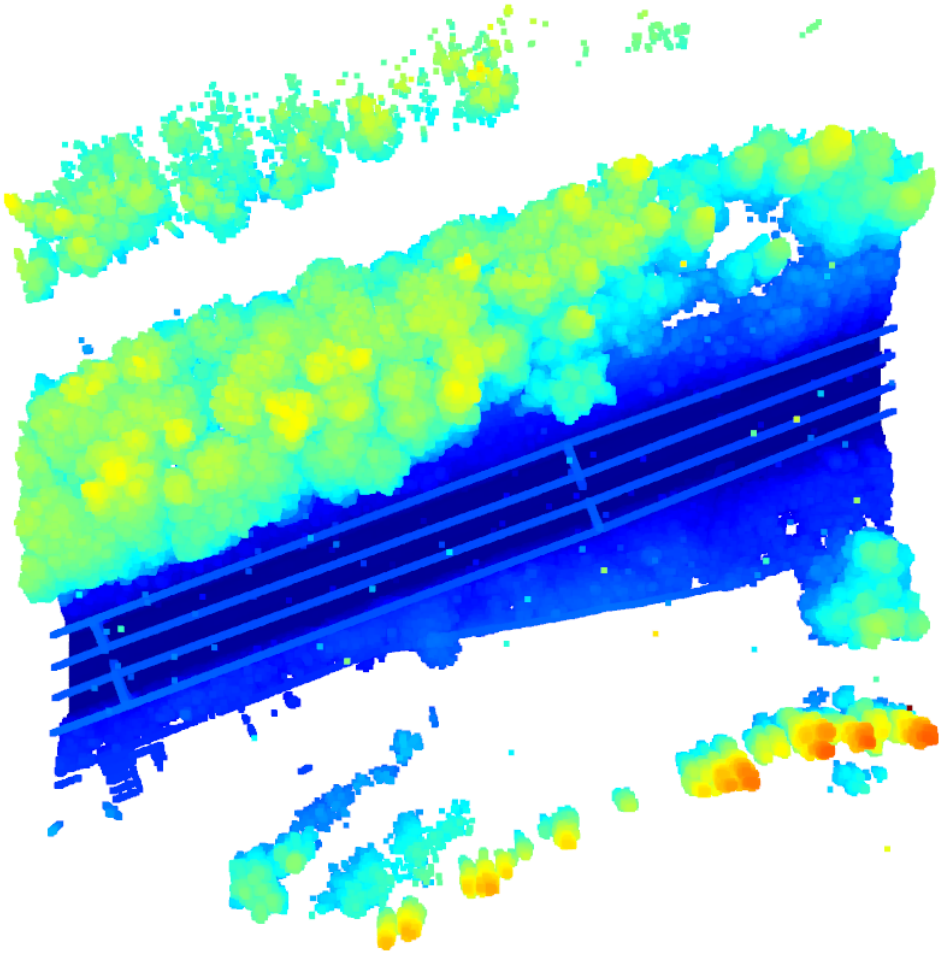


Figure 63: Point cloud data (coloured, October 2019) of tile (340500, 674200).

```
>>> pcd2
[PointCloud with 54007138 points.]
```

```
>>> pcd3 = pcdh.view_pcd_example(xy_tile, '202004', greyscale=True, width=1000)
```

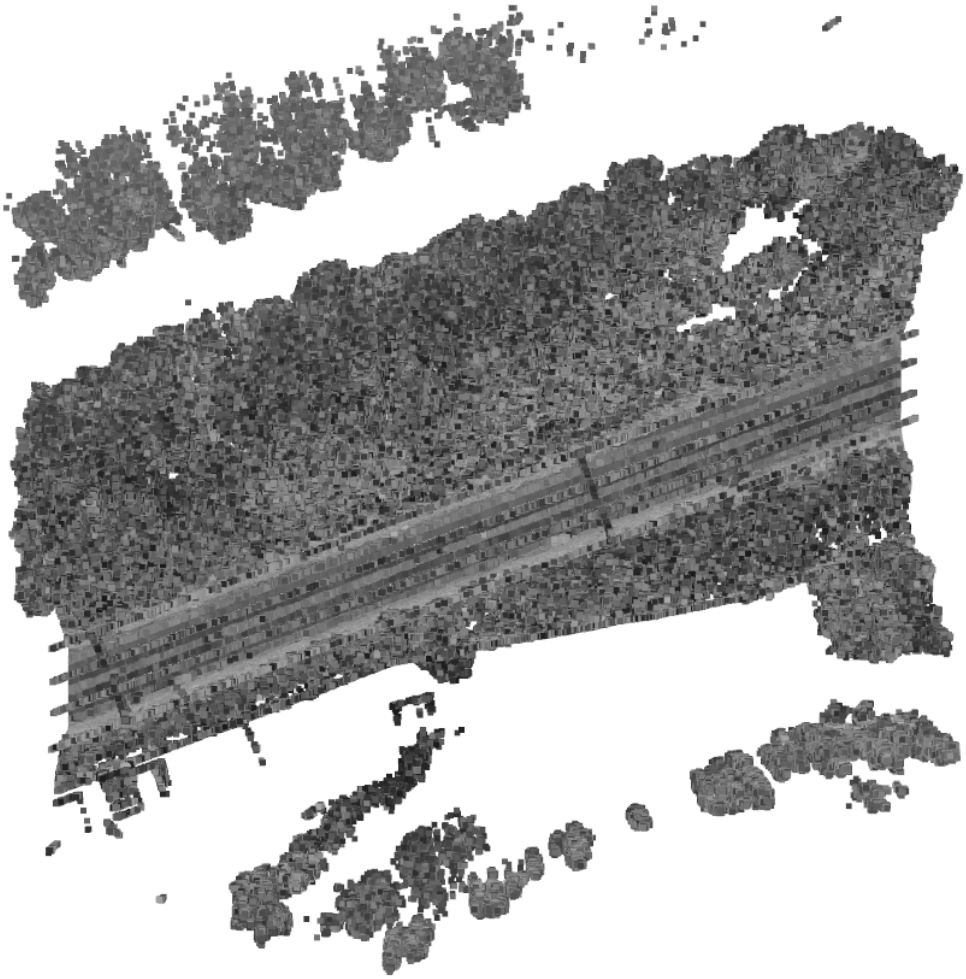


Figure 64: Point cloud data (greyscale, April 2020) of tile (340500, 674200).

```
>>> pcd3
[PointCloud with 20505100 points.]
```

```
>>> pcd4 = pcdh.view_pcd_example(xy_tile, pcd_dates='202004', width=1000)
```

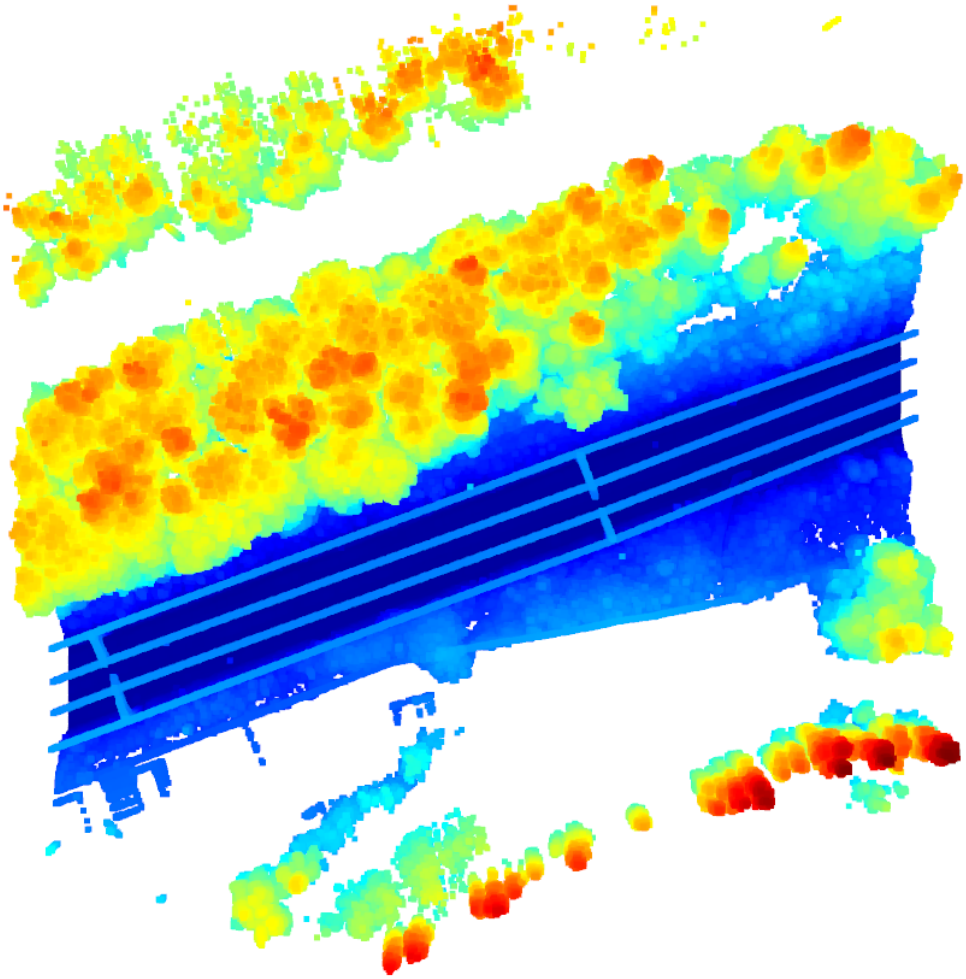


Figure 65: Point cloud data (coloured, April 2020) of tile (340500, 674200).

```
>>> pcd4
[PointCloud with 20505100 points.]
```

KRDZGear

class src.shaft.**KRDZGear**(*elr='ECM8'*, *db_instance=None*)

Further process the KRDZ data (within the point cloud category).

Parameters

- **elr** (*str*) – Engineer's Line Reference; defaults to 'ECM8'.
- **db_instance** (TrackFixityDB | *None*) – PostgreSQL database instance; defaults to None..

Variables

- **elr** (*str*) – Engineer's Line Reference.
- **trk** (Track) – Instance of the class *Track*.

Examples:

```
>>> from src.shaft import KRDZGear
>>> krdzg = KRDZGear()
>>> krdzg.NAME
"KRDZ data"
```

Attributes:

DIRECTIONS	Railway directions, including 'up' (being towards a major location) and 'down'.
ELEMENTS	Different parts of rail head, e.g. left/right top of rail or running edge.
NAME	Descriptive name of the class.

KRDZGear.DIRECTIONS

KRDZGear.**DIRECTIONS: list = ['Up', 'Down']**

Railway directions, including 'up' (being towards a major location) and 'down'.

KRDZGear.ELEMENTS

KRDZGear.**ELEMENTS: list = ['LeftTopOfRail', 'LeftRunningEdge', 'RightTopOfRail', 'RightRunningEdge', 'Centre']**

Different parts of rail head, e.g. left/right top of rail or running edge.

KRDZGear.NAME

KRDZGear.**NAME: str = 'KRDZ data'**

Descriptive name of the class.

Methods:

classify_krdz(tile_xy, pcd_date[, ...])	Classify KRDZ data by railway directions.
distinguish_between_up_and_down(xyz1, xyz2, ...)	Distinguish the (up and down) running directions of the two given arrays of rail track data.
gather_classified_krdz(pcd_date[, ...])	Collect together all classified KRDZ data for a given date.
get_adjusted_ref_line_par(ref_line_par, ...)	Adjust the position of a given reference line for clustering the original KRDZ data.
get_key_reference(tiles_convex_hull[, ...])	Get a main reference object for classifying the KRDZ data between up and down directions.
get_krdz_in_pcd_tile(xyz, tile_xy[, ...])	Find all points that are within a specified tile and data date from a given set of geographic coordinates (or point data).
get_pcd_tile_tracks_shp(trk_shp, tile_poly)	Get track shapefiles with respect to a given tile (for point cloud data).
get_reference_objects_for_krdz_clf(pcd_date)	Get three different reference objects for distinguishing the KRDZ data between up and down directions.
get_tiles_convex_hull(pcd_tiles[, as_array])	Get a representation of the smallest convex polygon containing all the tiles for the point cloud data.
get_tracks_shp_for_krdz_clf(tiles_convex_hull)	Get track shapefiles for clustering the KRDZ data.
get_tracks_shp_reference(trk_shp, tile_poly)	Get track shapefiles outside a buffer of a given tile (for point cloud data), used as a reference for clustering the KRDZ data within the tile.
import_classified_krdz([update, verbose])	Import the classified KRDZ rail head data into the project database.
load_classified_krdz([tile_xy, pcd_date, ...])	Load data of classified KRDZ rail head data from the project database.
load_pcd_krdz([tile_xy, pcd_date])	Get (X, Y, Z) coordinates of the left and right tops, as well as the running edges, of the rail heads from the KRDZ data.
view_classified_krdz(tile_xy, pcd_date[, ...])	View classified KRDZ data of the rail heads of point cloud data.
view_pcd_krdz(tile_xy, pcd_date[, ...])	Visualise original KRDZ data of the rail heads of point cloud data.

KRDZGear.classify_krdz

KRDZGear.**classify_krdz**(*tile_xy*, *pcd_date*, *ref_objects=None*, *update=False*, *verbose=True*, ***kwargs*)

Classify KRDZ data by railway directions.

Parameters

- **tile_xy** (*tuple | list | str*) – X and Y coordinates in reference to a tile for the point cloud data.
- **pcd_date** (*str | int*) – Date of the point cloud data.
- **ref_objects** (*tuple | None*) – Reference objects for identifying the running direction.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to `True`.
- **kwargs** – [Optional] parameters of sklearn.cluster.DBSCAN.

Returns

Coordinates of upside and downside rail heads, and associated labels.

Return type

dict

Examples:

```
>>> from src.shaft import KRDZGear
>>> krdzg = KRDZGear()
>>> # tile_xy, pcd_date = (340500, 674200), '201910'
>>> classified_krdz_201910 = krdzg.classify_krdz(
...     tile_xy=(340500, 674200), pcd_date='201910', n_jobs=-1)
>>> type(classified_krdz_201910)
dict
>>> len(classified_krdz_201910)
10
>>> list(classified_krdz_201910.keys())
['Up_LeftTopOfRail',
 'Up_LeftRunningEdge',
 'Up_RightTopOfRail',
 'Up_RightRunningEdge',
 'Up_Centre',
 'Down_RightTopOfRail',
 'Down_RightRunningEdge',
 'Down_LeftTopOfRail',
 'Down_LeftRunningEdge',
 'Down_Centre']
>>> ult_ls_201910 = classified_krdz_201910['Up_LeftTopOfRail']
>>> type(ult_ls_201910)
shapely.geometry.linestring.LineString
>>> ult_ls_201910.length
```

(continues on next page)

(continued from previous page)

```
105.9959864289052
>>> urt_ls_201910 = classified_krdz_201910['Up_RightTopOfRail']
>>> urt_ls_201910.length
105.99667584951499
>>> # tile_xy, pcd_date = (340500, 674200), '202004'
>>> classified_krdz_202004 = krdzg.classify_krdz(
...     tile_xy=(340500, 674200), pcd_date='202004', n_jobs=-1)
>>> ult_ls_202004 = classified_krdz_202004['Up_LeftTopOfRail']
>>> ult_ls_202004.length
105.99856695815154
```

See also:

For detailed illustration of the method, see debugging.ipynb.

KRDZGear.distinguish_between_up_and_down

KRDZGear.**distinguish_between_up_and_down**(*xyz1*, *xyz2*, *tiles_convex_hull*, *ref_line_par*, *tile_poly*, *tile_trk_shp_ref*)

Distinguish the (up and down) running directions of the two given arrays of rail track data.

The rail track data could be KRDZ data and track shapefile data.

Parameters

- **xyz1** (*numpy.ndarray*) – One array of rail track data.
- **xyz2** (*numpy.ndarray*) – Another array of rail track data.
- **tiles_convex_hull** (*Polygon*) – Convex hull of all tiles for the point cloud data.
- **ref_line_par** (*LineString*) – A reference line that is parallel to a pre-specified one.
- **tile_poly** (*Polygon*) – A tile.
- **tile_trk_shp_ref** (*LineString*) – A subsection of track shapefile used for reference.

Returns

Data of rail track in the up direction and that in the down direction.

Return type

tuple

Examples:

```
>>> from src.shaft import KRDZGear
>>> from src.utils import get_tile_xy
>>> from sklearn.cluster import DBSCAN
>>> from pyhelpers.geom import find_shortest_path
```

(continues on next page)

(continued from previous page)

```
>>> import numpy as np
>>> krdzg = KRDZGear()
>>> pcd_date, tile_xy = '202004', (399600, 654100)
>>> tile_x, tile_y = get_tile_xy(tile_xy=tile_xy)
>>> # KRDZ data
>>> krdz_xyz = krdzg.load_pcd_krdz(tile_xy=(tile_x, tile_y), pcd_date=pcd_date)
>>> dat = krdz_xyz['LeftTopOfRail']
>>> clusters = DBSCAN(eps=2, min_samples=2, algorithm='brute').fit(dat)
>>> labels = np.unique(clusters.labels_)
>>> xyz_1 = find_shortest_path(dat[clusters.labels_ == labels[0]])
>>> xyz_2 = find_shortest_path(dat[clusters.labels_ != labels[0]])
>>> len(xyz_1)
100
>>> len(xyz_2)
100
>>> ch, ref_lp, trks = krdzg.get_reference_objects_for_krdz_clf(pcd_date, par_
 ↪dist=5000)
>>> tile_polygon = krdzg.get_pcd_tile_polygon(tile_xy=(tile_x, tile_y), as_
 ↪geom=True)
>>> _, tile_tracks_shp_ref = krdzg.get_tracks_shp_reference(trks, tile_polygon)
>>> up_xyz, down_xyz = krdzg.distinguish_between_up_and_down(
...     xyz1=xyz_1, xyz2=xyz_2, tiles_convex_hull=ch, ref_line_par=ref_lp,
...     tile_poly=tile_polygon, tile_trk_shp_ref=tile_tracks_shp_ref)
>>> type(up_xyz)
shapely.geometry.linestring.LineString
>>> print(up_xyz.wkt)
LINESTRING Z (399641.403 654100.317 37.556, 399641.445 654101.316 37.558,␣
 ↪399641.48...
>>> type(down_xyz)
shapely.geometry.linestring.LineString
>>> print(down_xyz.wkt)
LINESTRING Z (399645.168 654199.431 37.987, 399645.203 654198.433 37.983,␣
 ↪399645.23...
```

See also:

For detailed illustration of the method, see debugging.ipynb.

KRDZGear.gather_classified_krdz

KRDZGear.**gather_classified_krdz**(*pcd_date*, *set_index=None*, *update=False*, *verbose=False*, ***kwargs*)

Collect together all classified KRDZ data for a given date.

Parameters

- **pcd_date** (*str | int*) – Date of the point cloud data.
- **set_index** (*bool | list | None*) – Whether to set an index (or indexes), or what to be set as an index (or indexes); defaults to None.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to False.

- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to `True`.

Returns
KRDZ rail head data (classified by up and down directions).

Return type
pandas.DataFrame

Examples:

```
>>> from src.shaft import KRDZGear
>>> krdzg = KRDZGear()
>>> # -- October 2019 ----------------------------------
>>> # classified_krdz_201910 = tm.gather_classified_krdz(
... #    pcd_date='201910', update=True, verbose=True, n_jobs=-1)
>>> classified_krdz_201910 = krdzg.gather_classified_krdz(pcd_date='201910')
>>> classified_krdz_201910
      Year  ...                                           geometry
0     2019  ...  LINESTRING Z (340199.193 674110.113 33.165, 34...
1     2019  ...  LINESTRING Z (340199.18 674110.147 33.15, 3401...
2     2019  ...  LINESTRING Z (340199.601 674111.877 33.163, 34...
3     2019  ...  LINESTRING Z (340199.613 674111.843 33.148, 34...
4     2019  ...  LINESTRING Z (340199.866 674111.17 33.163, 340...
...    ...  ...                                             ...
9807  2019  ...  LINESTRING Z (340708.001 674299.684 32.27, 340...
9808  2019  ...  LINESTRING Z (340707.987 674299.719 32.253, 34...
9809  2019  ...  LINESTRING Z (340703.726 674299.666 32.214, 34...
9810  2019  ...  LINESTRING Z (340704.674 674299.985 32.201, 34...
9811  2019  ...  LINESTRING Z (340705.871 674299.677 32.242, 34...
[9812 rows x 7 columns]
>>> # -- April 2020 ------------------------------------
>>> # classified_krdz_202004 = tm.gather_classified_krdz(
... #    pcd_date='202004', update=True, verbose=True, n_jobs=-1)
>>> classified_krdz_202004 = krdzg.gather_classified_krdz(pcd_date='202004')
>>> classified_krdz_202004
      Year  ...                                           geometry
0     2020  ...  LINESTRING Z (340132.744 674085.354 33.344, 34...
1     2020  ...  LINESTRING Z (340132.731 674085.389 33.329, 34...
2     2020  ...  LINESTRING Z (340132.196 674086.761 33.347, 34...
3     2020  ...  LINESTRING Z (340132.21 674086.727 33.332, 340...
4     2020  ...  LINESTRING Z (340132.47 674086.058 33.345, 340...
...    ...  ...                                             ...
9812  2020  ...  LINESTRING Z (399600.071 654486.99 39.58, 3996...
9813  2020  ...  LINESTRING Z (399600.106 654486.999 39.567, 39...
9814  2020  ...  LINESTRING Z (399600.129 654492.216 39.682, 39...
9815  2020  ...  LINESTRING Z (399600.096 654492.205 39.664, 39...
9816  2020  ...  LINESTRING Z (399600.241 654489.12 39.628, 399...
[9817 rows x 7 columns]
```

KRDZGear.get_adjusted_ref_line_par

static KRDZGear.**get_adjusted_ref_line_par**(*ref_line_par*, *tile_poly*, *aux_ref=None*, *centroid=False*)

Adjust the position of a given reference line for clustering the original KRDZ data.

Parameters

- **ref_line_par** (*LineString*) – A reference line that is parallel to a pre-specified one.
- **tile_poly** (*Polygon*) – A tile.
- **aux_ref** (*LineString | None*) – An auxiliary reference line; defaults to None.
- **centroid** (*bool*) – Whether to use the centroid of the adjusted reference line; defaults to False.

Returns

Adjusted reference location based on the given reference line.

Return type

LineString | Point

Examples:

```
>>> from src.shaft import KRDZGear
>>> krdzg = KRDZGear()
>>> ref_objects = krdzg.get_reference_objects_for_krdz_clf('202004', par_
↪dist=5000)
>>> tiles_convex_hull, par_ref_ls, _ = ref_objects
>>> type(par_ref_ls)
shapely.geometry.linestring.LineString
>>> print(par_ref_ls.wkt)
LINESTRING (397656.7781152923 648677.7312827818, 338456.7781152923 669277.
↪7312827818)
>>> tile_x_y = (340100, 674000)
>>> tile_polygon = krdzg.get_pcd_tile_polygon(tile_xy=tile_x_y)
>>> type(tile_polygon)
shapely.geometry.polygon.Polygon
>>> print(tile_polygon.wkt)
POLYGON ((340100 674000, 340100 674100, 340200 674100, 340200 674000, 340100␣
↪674000))
>>> par_ref_ls_adj = krdzg.get_adjusted_ref_line_par(par_ref_ls, tile_poly=tile_
↪polygon)
```

Illustration:

```
import matplotlib.pyplot as plt
import numpy as np
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg')
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(11, 5), constrained_layout=True)
ax = fig.add_subplot()
ax.set_aspect('equal', adjustable='box')

tch_arr = np.array(tiles_convex_hull.exterior.coords)
ax.plot(tch_arr[:, 0], tch_arr[:, 1], zorder=3, label='Convex hull for all tiles
↪')

ref_line = krdzg.get_key_reference(tiles_convex_hull)
rl_arr = np.array(ref_line.coords)
ax.plot(rl_arr[:, 0], rl_arr[:, 1], lw=5, label='Ref line')

tp_arr = np.array(tile_polygon.exterior.coords)
ax.plot(tp_arr[:, 0], tp_arr[:, 1], label='Tile of (340100, 674000)')

prl_arr = np.array(par_ref_ls.coords)
ax.plot(prl_arr[:, 0], prl_arr[:, 1], lw=5, zorder=2, label='Ref line␣
↪(Paralleled)')

aprl_arr = np.array(par_ref_ls_adj.coords)
ax.plot(aprl_arr[:, 0], aprl_arr[:, 1], lw=3, zorder=3, label='Ref line␣
↪(Adjusted)')

ax.legend()

ax.set_xlabel('Easting', fontsize=13, labelpad=5)
ax.set_ylabel('Northing', fontsize=13, labelpad=5)

# from pyhelpers.store import save_figure
#
# fig_pathname = "docs/source/_images/krdzg_get_adj_ref_ls_par_demo"
# save_figure(fig, f"{fig_pathname}.svg", verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", verbose=True)
#
# fig_pathname = "docs/source/_images/krdzg_get_adj_ref_ls_par_demo_zoomed_in"
# save_figure(fig, f"{fig_pathname}.svg", verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", verbose=True)
```

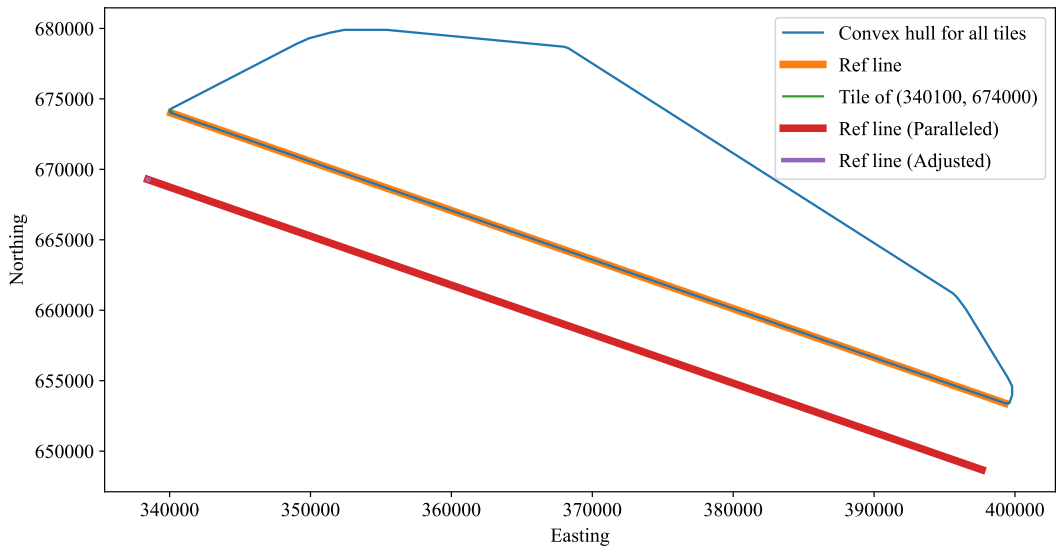


Figure 66: Adjusted reference line for classifying KRDZ data in Tile (340100, 674000).

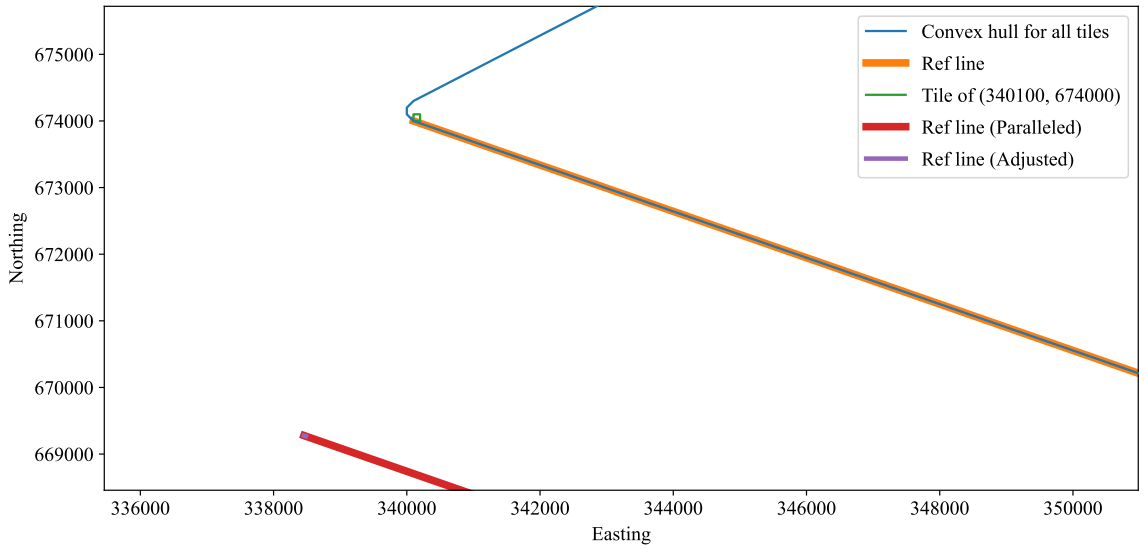


Figure 67: (Zoomed-in) Adjusted reference line for classifying KRDZ data in Tile (340100, 674000).

KRDZGear.get_key_reference

static KRDZGear.**get_key_reference**(*tiles_convex_hull*, *obj_attr=None*, *as_array=False*)

Get a main reference object for classifying the KRDZ data between up and down directions.

Parameters

- **tiles_convex_hull** (*Polygon*) – Convex hull of the tiles for the point cloud data.

- **obj_attr** (*str | None*) – Attribute of the reference object; options include `'centroid'` (for the centroid of the object) and `'rep'` (for the representative point of the object); when `obj_attr=None` (default), the object remains its default type as a linestring.
- **as_array** (*bool*) – Whether to convert the geometry object to array type; defaults to `False`.

Returns

A reference point for identifying up and down directions.

Return type

LineString | Point | numpy.ndarray

Examples:

```
>>> from src.shaft import KRDZGear
>>> krdzg = KRDZGear()
>>> pcd_tiles_metadata_201910 = krdzg.load_tiles(pcd_date='201910')
>>> pcd_tiles_metadata_201910.head()
   Year  ...                                        Tile_LonLat
0  2019  ...  POLYGON ((-2.960910975517333 55.95617100494877...
1  2019  ...  POLYGON ((-2.960888715229412 55.95527265723062...
2  2019  ...  POLYGON ((-2.012726922542862 55.77482976892926...
3  2019  ...  POLYGON ((-2.009538521884446 55.77393156353337...
4  2019  ...  POLYGON ((-2.01113257284158 55.77393142614289,...
[5 rows x 6 columns]
>>> pcd_tiles_201910 = pcd_tiles_metadata_201910['Tile_XY']
>>> pcd_tiles_201910.head()
0    POLYGON ((340100 674100, 340100 674200, 340200...
1    POLYGON ((340100 674000, 340100 674100, 340200...
2    POLYGON ((399300 653500, 399300 653600, 399400...
3    POLYGON ((399500 653400, 399500 653500, 399600...
4    POLYGON ((399400 653400, 399400 653500, 399500...
Name: Tile_XY, dtype: object
>>> pcd_tiles_convex_hull = krdzg.get_tiles_convex_hull(pcd_tiles_201910)
>>> print(pcd_tiles_convex_hull.wkt)
POLYGON ((399300 653400, 340100 674000, 340100 674300, 348300 678500, 349300␣
 ↪679000, ...
>>> ref_line = krdzg.get_key_reference(pcd_tiles_convex_hull)
>>> type(ref_line)
shapely.geometry.linestring.LineString
>>> print(ref_line.wkt)
LINESTRING (399300 653400, 340100 674000)
>>> ref_pt = krdzg.get_key_reference(pcd_tiles_convex_hull, obj_attr='centroid')
>>> type(ref_pt)
shapely.geometry.point.Point
>>> print(ref_pt.wkt)
POINT (370139.9108845997 668965.4040025512)
>>> ref_rep = krdzg.get_key_reference(pcd_tiles_convex_hull, obj_attr='rep')
>>> type(ref_rep)
shapely.geometry.point.Point
>>> print(ref_rep.wkt)
POINT (372007.0360450842 667600)
```

Illustration:

```
import matplotlib.pyplot as plt
import numpy as np
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg')

fig = plt.figure(constrained_layout=True, figsize=(11, 5))
ax = fig.add_subplot()
ax.set_aspect(aspect='equal', adjustable='box')

colours = plt.get_cmap('tab10').colors

convex_hull_arr = np.array(pcd_tiles_convex_hull.exterior.coords)
ax.plot(
    convex_hull_arr[:, 0], convex_hull_arr[:, 1], color=colours[0], zorder=2,
    label='Convex hull for all tiles\n' \
          '(The area based on which the key reference objects are determined).')

ref_line_coords = np.array(ref_line.coords)
ax.plot(
    ref_line_coords[:, 0], ref_line_coords[:, 1], color=colours[1], linewidth=5,
    label='Ref LineString', zorder=1)
ax.scatter(ref_pt.x, ref_pt.y, color=colours[2], label='Ref Centroid')
ax.scatter(ref_rep.x, ref_rep.y, color=colours[3], label='Ref Rep')

ax.legend()

ax.set_xlabel('Easting', fontsize=13, labelpad=5)
ax.set_ylabel('Northing', fontsize=13, labelpad=5)

# from pyhelpers.store import save_figure
#
# fig_pathname = "docs/source/_images/krdzg_get_key_reference_demo"
# save_figure(fig, f"{fig_pathname}.svg", verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", verbose=True)
```

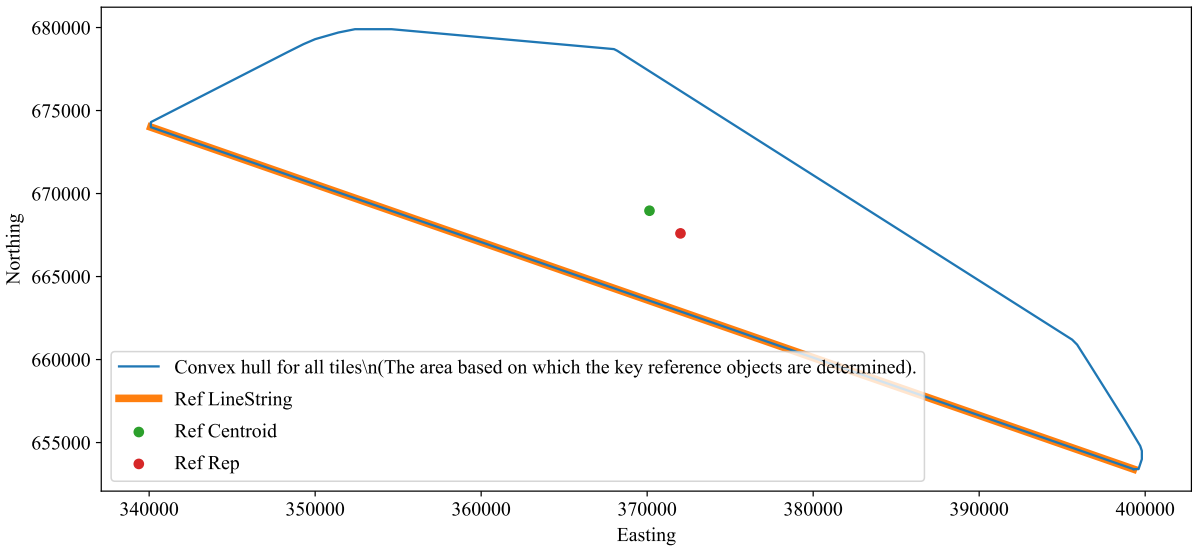


Figure 68: Reference objects for classifying KRDZ data.

KRDZGear.get_krdz_in_pcd_tile

KRDZGear.**get_krdz_in_pcd_tile**(*xyz*, *tile_xy*, *pcd_date=None*, *radius=0.0*)

Find all points that are within a specified tile and data date from a given set of geographic coordinates (or point data).

Parameters

- **xyz** (*numpy.ndarray | LineString*) – Geographic coordinates (or point data).
- **tile_xy** (*tuple | list | str*) – X and Y coordinates in reference to a tile for the point cloud data.
- **pcd_date** (*str | int | None*) – Date of the point cloud data; defaults to None.
- **radius** (*float*) – Radius; defaults to 0.0.

Returns

All data within the given tile_xy and data date pcd_date.

Return type

numpy.ndarray

Examples:

```
>>> from src.shaft import KRDZGear
>>> from pyhelpers.settings import np_preferences
>>> np_preferences()
>>> krdzg = KRDZGear()
>>> # Get KRDZ rail head data
>>> krdz_data = krdzg.load_krdz(pcd_dates='201910')
>>> krdz_data
        Year  Month  ...  zRightRunningEdge  zRightTopOfRail
0       2019     10  ...             33.341           33.356
1       2019     10  ...             33.335           33.351
2       2019     10  ...             33.331           33.346
3       2019     10  ...             33.327           33.343
4       2019     10  ...             33.324           33.340
...      ...    ...  ...                ...              ...
152616  2019     10  ...             33.316           33.332
152617  2019     10  ...             33.319           33.335
152618  2019     10  ...             33.322           33.337
152619  2019     10  ...             33.324           33.339
152620  2019     10  ...             33.327           33.342
[152621 rows x 29 columns]
>>> left_top_cols = ['xLeftTopOfRail', 'yLeftTopOfRail', 'zLeftTopOfRail']
>>> left_top_201910 = krdz_data[left_top_cols].to_numpy()
>>> left_top_201910.shape
(152621, 3)
>>> lt_201910 = krdzg.get_krdz_in_pcd_tile(left_top_201910, tile_xy=(340500,␣
↪674200))
>>> lt_201910.shape
(214, 3)
```

KRDZGear.get_pcd_tile_tracks_shp

static KRDZGear.**get_pcd_tile_tracks_shp**(*trk_shp*, *tile_poly*)

Get track shapefiles with respect to a given tile (for point cloud data).

Parameters

- **trk_shp** (*MultiLineString*) – A number of geometry objects of track lines.
- **tile_poly** (*Polygon*) – A tile.

Returns

Track lines that intersect or are near the given tile.

Return type

MultiLineString

Examples:

```
>>> from src.shaft import KRDZGear
>>> from shapely.geometry import MultiLineString
>>> krdzg = KRDZGear()
>>> trk_shp_data = krdzg.trk.load_tracks_shp(elr=['ECM7', 'ECM8'])
>>> track_shp = MultiLineString(trk_shp_data['geometry'].to_list())
>>> type(track_shp)
shapely.geometry.multilinestring.MultiLineString
>>> len(track_shp.geoms)
440
>>> tile_xy = (380600, 665400)
>>> tile_polygon = krdzg.get_pcd_tile_polygon(tile_xy=tile_xy)
>>> type(tile_polygon)
shapely.geometry.polygon.Polygon
>>> print(tile_polygon.bounds)
(380600.0, 665400.0, 380700.0, 665500.0)
>>> track_lines = krdzg.get_pcd_tile_tracks_shp(track_shp, tile_poly=tile_
↪polygon)
>>> type(track_lines)
shapely.geometry.multilinestring.MultiLineString
>>> len(track_lines.geoms)
2
```

Illustration:

```
import matplotlib.pyplot as plt
import numpy as np
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg')

colours = plt.get_cmap('tab10').colors

fig = plt.figure(figsize=(6, 6), constrained_layout=True)
ax = fig.add_subplot(aspect='equal', adjustable='box')

tile_poly_coords = np.array(tile_polygon.exterior.coords)
ax.plot(
```

(continues on next page)

(continued from previous page)

```
    tile_poly_coords[:, 0], tile_poly_coords[:, 1], color=colours[0],
    label='Tile (380600, 665400)')

for ls in track_shp.geoms:
    ls_coords = np.array(ls.coords)
    ax.plot(ls_coords[:, 0], ls_coords[:, 1], color=colours[1])
ax.plot([], [], color=colours[1], label='Track shapefiles')

for ls in track_lines.geoms:
    ls_coords = np.array(ls.coords)
    ax.plot(ls_coords[:, 0], ls_coords[:, 1], color=colours[2], linewidth=3)
ax.plot(
    [], [], color=colours[2], linewidth=3,
    label='Track shapefiles w.r.t. the Tile (380600, 665400)')

xmi, xma, ymi, yma = map(lambda x: int(x//10000*10000), ax.get_xlim() + ax.get_
↪ylim())
ax.xaxis.set_ticks(range(xmi, xma, 20000))
ax.yaxis.set_ticks(range(ymi, yma, 20000))

ax.legend()

ax.set_xlabel('Easting', fontsize=13, labelpad=5)
ax.set_ylabel('Northing', fontsize=13, labelpad=5)

# ax.xaxis.set_ticks([380600, 380700])
# ax.yaxis.set_ticks([665400, 665500])

# from pyhelpers.store import save_figure
#
# fig_pathname = "docs/source/_images/krdzg_get_trk_shp_for_pcd_tile_demo"
# save_figure(fig, f"{fig_pathname}.svg", verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", verbose=True)
#
# fig_pathname = "docs/source/_images/krdzg_get_trk_shp_for_pcd_tile_demo_
↪zoomed_in"
# save_figure(fig, f"{fig_pathname}.svg", verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", verbose=True)
```

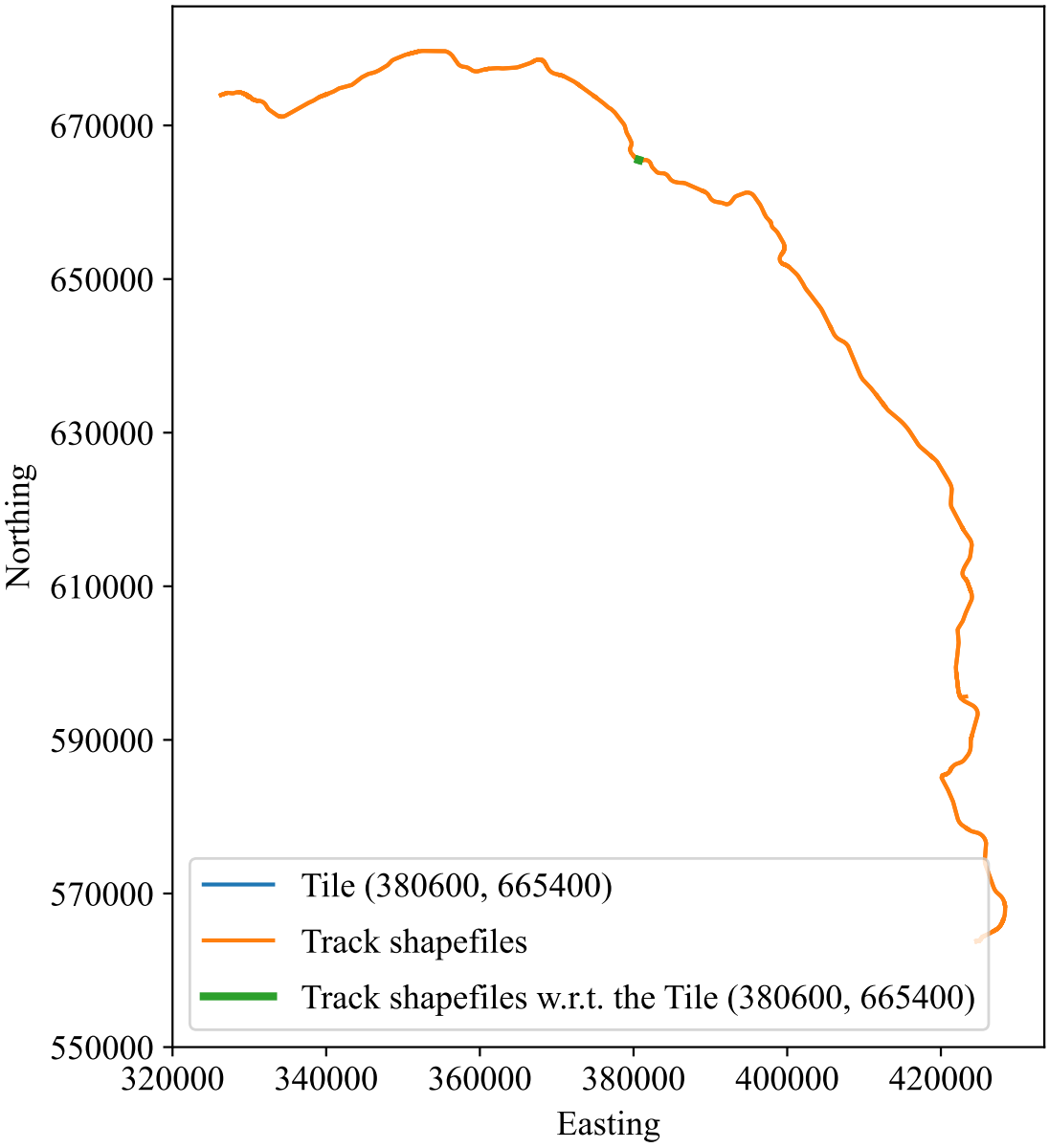


Figure 69: The track shapefile with regard to the Tile (380600, 665400).

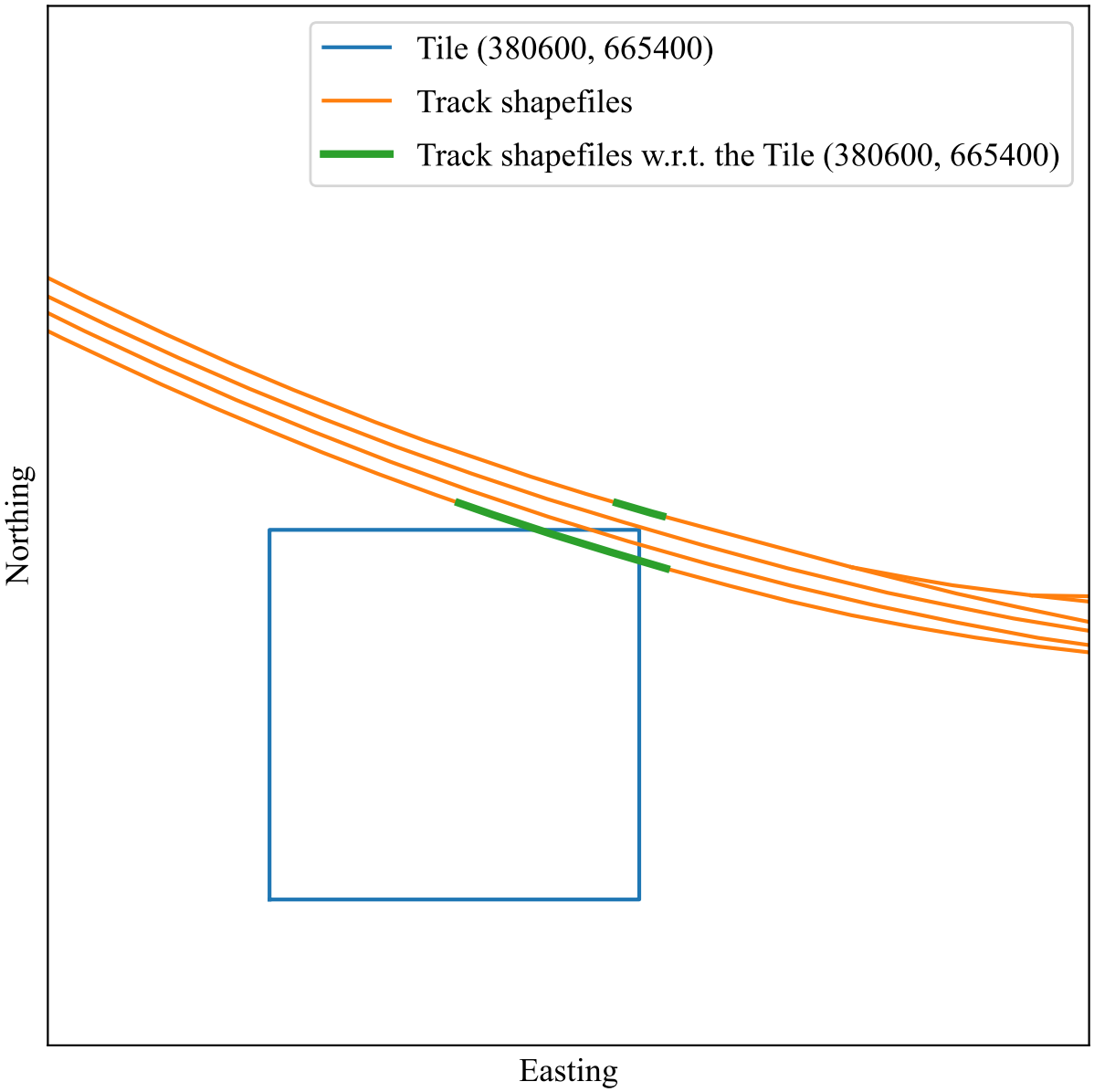


Figure 70: (Zoomed-in) The track shapefile with regard to the Tile (380600, 665400).

KRDZGear.get_reference_objects_for_krdz_clf

KRDZGear.**get_reference_objects_for_krdz_clf**(*pcd_date*, *par_dist=5000*, *ret_tile_names=False*)

Get three different reference objects for distinguishing the KRDZ data between up and down directions.

Parameters

- **pcd_date** (*str | int*) – Date of point cloud data.
- **par_dist** (*int | float*) – Distance to parallel offset the key reference line; defaults to 5000.

- **ret_tile_names** (*bool*) – Whether to return tile names as well; defaults to `False`.

Returns
Reference objects for distinguishing the KRDZ data between up and down directions.

Return type
tuple

Examples:

```
>>> from src.shaft import KRDZGear
>>> krdzg = KRDZGear()
>>> ref_objects = krdzg.get_reference_objects_for_krdz_clf(pcd_date='202004')
>>> convex_hull, ref_line_paralleled, track_shp = ref_objects
>>> type(convex_hull)
shapely.geometry.polygon.Polygon
>>> type(ref_line_paralleled)
shapely.geometry.linestring.LineString
>>> type(track_shp)
shapely.geometry.multilinestring.MultiLineString
```

Illustration:

```
import matplotlib.pyplot as plt
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg', font_name='Times New Roman')

fig = plt.figure(figsize=(11, 5), constrained_layout=True)
ax = fig.add_subplot()
ax.set_aspect('equal', adjustable='box')

colours = plt.get_cmap('tab10').colors

xs, ys = convex_hull.exterior.coords.xy
ax.plot(xs, ys, color=colours[1], label='Convex hull for all tiles')

xs, ys = ref_line_paralleled.coords.xy
ax.plot(xs, ys, color=colours[0], label='Ref line (paralleled)')

for trk_ls in track_shp.geoms:
    xs, ys = trk_ls.coords.xy
    ax.plot(xs, ys, color=colours[2])
ax.plot([], [], color=colours[2], label='Track shapefiles')

ax.legend()

ax.set_xlabel('Easting', fontsize=13, labelpad=5)
ax.set_ylabel('Northing', fontsize=13, labelpad=5)

# from pyhelpers.store import save_figure
# fig_pathname = "docs/source/_images/krdzg_get_ref_obj_for_krdz_clf_demo"
# save_figure(fig, f"{fig_pathname}.svg", verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", verbose=True)
```

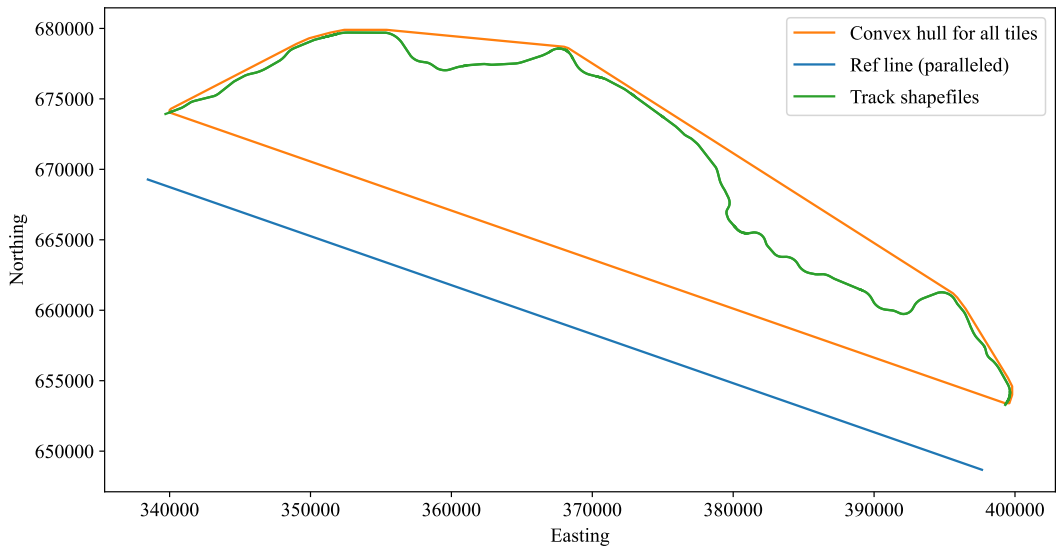


Figure 71: Main reference objects for clustering the KRDZ data.

KRDZGear.get_tiles_convex_hull

static KRDZGear.**get_tiles_convex_hull**(*pcd_tiles*, *as_array=False*)

Get a representation of the smallest convex polygon containing all the tiles for the point cloud data.

Parameters

- **pcd_tiles** (*numpy.ndarray | pandas.Series | pandas.DataFrame*) – The tiles for the point cloud data.
- **as_array** (*bool*) – Whether to return the polygon as an array of vertices; defaults to `False`.

Returns

The smallest convex polygon containing all the tiles for the point cloud data.

Return type

Polygon | numpy.ndarray

Examples:

```
>>> from src.shaft import KRDZGear
>>> krdzg = KRDZGear()
>>> pcd_tiles_metadata_201910 = krdzg.load_tiles(pcd_date='201910')
>>> pcd_tiles_metadata_201910.head()
   Year  ...                                          Tile_LonLat
0  2019  ...  POLYGON ((-2.960910975517333 55.95617100494877...
1  2019  ...  POLYGON ((-2.960888715229412 55.95527265723062...
2  2019  ...  POLYGON ((-2.012726922542862 55.77482976892926...
3  2019  ...  POLYGON ((-2.009538521884446 55.77393156353337...
4  2019  ...  POLYGON ((-2.01113257284158 55.77393142614289,...
[5 rows x 6 columns]
```

(continues on next page)

(continued from previous page)

```
>>> pcd_tiles_201910 = pcd_tiles_metadata_201910['Tile_XY']
>>> pcd_tiles_201910.head()
0    POLYGON ((340100 674100, 340100 674200, 340200...
1    POLYGON ((340100 674000, 340100 674100, 340200...
2    POLYGON ((399300 653500, 399300 653600, 399400...
3    POLYGON ((399500 653400, 399500 653500, 399600...
4    POLYGON ((399400 653400, 399400 653500, 399500...
Name: Tile_XY, dtype: object
>>> convex_hull = krdzg.get_tiles_convex_hull(pcd_tiles=pcd_tiles_201910)
>>> type(convex_hull)
shapely.geometry.polygon.Polygon
>>> print(convex_hull.wkt)
POLYGON ((399300 653400, 340100 674000, 340100 674300, 348300 678500, 349300␣
 ↪679000, ...
>>> convex_hull_arr = krdzg.get_tiles_convex_hull(pcd_tiles_201910, as_
 ↪array=True)
>>> type(convex_hull_arr)
numpy.ndarray
>>> convex_hull_arr.shape
(20, 2)
```

Illustration:

```
import matplotlib.pyplot as plt
import numpy as np
import shapely.wkt
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg')

fig = plt.figure(constrained_layout=True, figsize=(11, 5))
ax = fig.add_subplot()
ax.set_aspect(aspect='equal', adjustable='box')

colours = plt.get_cmap('tab10').colors

for tile in pcd_tiles_201910.map(shapely.wkt.loads):
    xs, ys = zip(*tile.exterior.coords)
    ax.plot(xs, ys, color=colours[0])
ax.scatter([], [], marker='s', facecolors='none', edgecolors=colours[0], label=
 ↪'Tile')

ch_xs, ch_ys = convex_hull_arr[:, 0], convex_hull_arr[:, 1]
ax.plot(ch_xs, ch_ys, color=colours[1], label='Convex hull for the tiles')

ax.legend()

xmi, xma, ymi, yma = map(lambda x: int(x//10000*10000), ax.get_xlim() + ax.get_
 ↪ylim())
ax.xaxis.set_ticks(range(xmi + 10000, xma + 10000, 10000))
ax.yaxis.set_ticks(range(ymi + 5000, yma + 5000, 5000))

ax.set_xlabel('Easting', fontsize=13, labelpad=5)
ax.set_ylabel('Northing', fontsize=13, labelpad=5)

# from pyhelpers.store import save_figure
```

(continues on next page)

(continued from previous page)

```
#
# fig_pathname = "docs/source/_images/krdzg_get_tiles_convex_hull_demo"
# save_figure(fig, f"{fig_pathname}.svg", verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", verbose=True)
#
# fig_pathname = "docs/source/_images/krdzg_get_tiles_convex_hull_demo_zoomed_in
↪"
# save_figure(fig, f"{fig_pathname}.svg", verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", verbose=True)
```

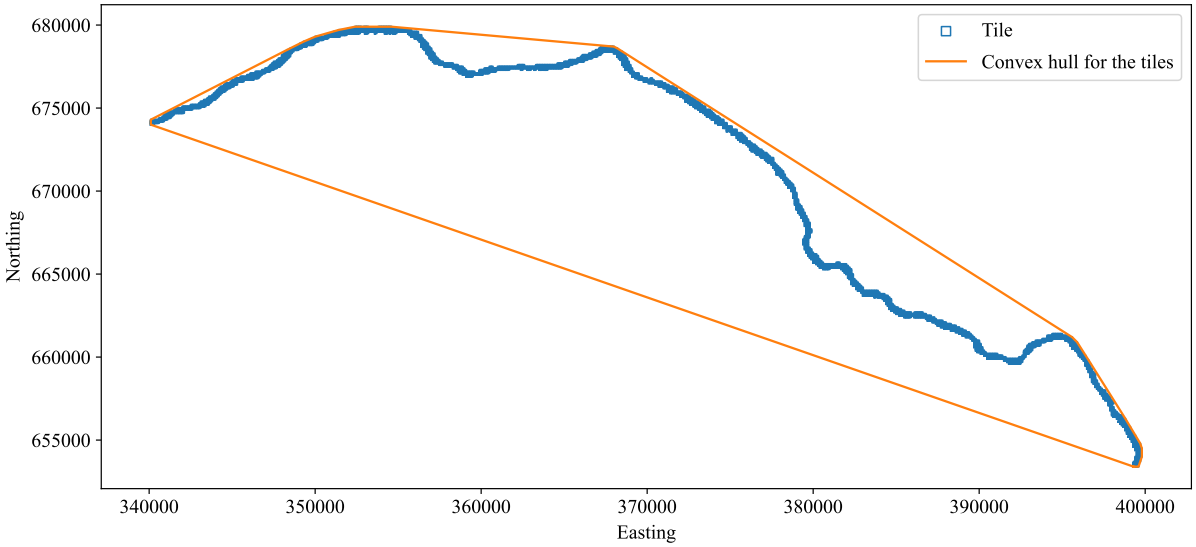


Figure 72: Convex hull for all tiles for the point cloud data.

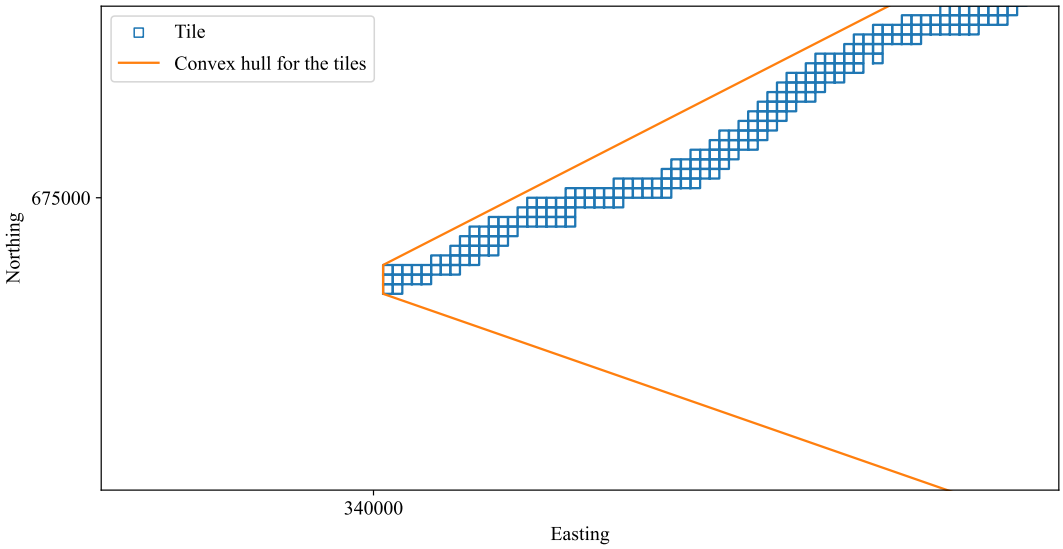


Figure 73: (Zoomed-in) Convex hull for all tiles for the point cloud data.

KRDZGear.get_tracks_shp_for_krdz_clf

KRDZGear.**get_tracks_shp_for_krdz_clf**(*tiles_convex_hull*, *tile_poly=None*, ***kwargs*)

Get track shapefiles for clustering the KRDZ data.

Parameters

- **tiles_convex_hull** (*Polygon*) – Convex hull of all tiles for the point cloud data
- **tile_poly** (*Polygon | None*) – A tile; defaults to None.
- **kwargs** – [Optional] parameters of the method *load_tracks_shp()*.

Returns

Track lines that are used for identifying KRDZ data between up and down directions.

Return type

MultiLineString

Examples:

```
>>> from src.shaft import KRDZGear
>>> from shapely.geometry import MultiLineString
>>> krdzg = KRDZGear()
>>> convex_hull = krdzg.get_tiles_convex_hull(krdzg.load_tiles()['Tile_XY'])
>>> track_shp = krdzg.get_tracks_shp_for_krdz_clf(tiles_convex_hull=convex_hull)
>>> type(track_shp)
shapely.geometry.multilinestring.MultiLineString
>>> len(track_shp.geoms)
102
```

Illustration:

```
import matplotlib.pyplot as plt
import numpy as np
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg')

fig = plt.figure(figsize=(11, 5), constrained_layout=True)
ax = fig.add_subplot()
ax.set_aspect('equal', adjustable='box')

colours = plt.get_cmap('tab10').colors

ch_arr = np.array(convex_hull.exterior.coords)
ax.plot(ch_arr[:, 0], ch_arr[:, 1], color=colours[1], label='Convex hull for␣
↪all tiles')

for trk_ls in track_shp.geoms:
    trk_xs, trk_xy = trk_ls.coords.xy
    ax.plot(trk_xs, trk_xy, color=colours[2])
```

(continues on next page)

(continued from previous page)

```
ax.plot([], [], color=colours[2], label='Track shapefiles for clustering the␣
↪KRDZ data')

ax.legend()

ax.set_xlabel('Easting', fontsize=13, labelpad=5)
ax.set_ylabel('Northing', fontsize=13, labelpad=5)

# from pyhelpers.store import save_figure
# fig_pathname = "docs/source/_images/krdzg_get_trk_shp_for_krdz_clf_demo"
# save_figure(fig, f"{fig_pathname}.svg", verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", verbose=True)
```

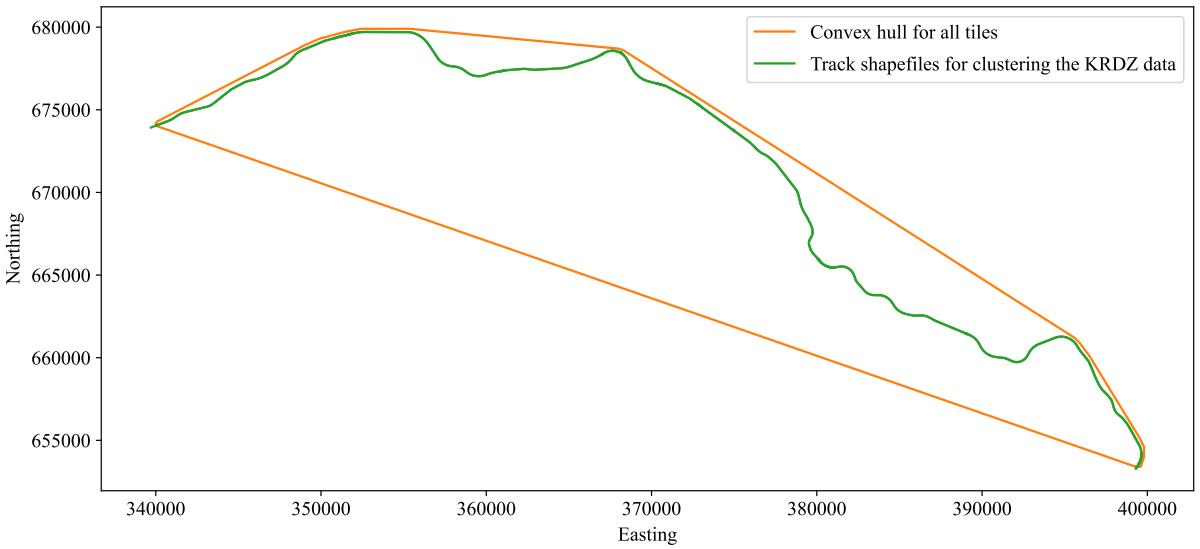


Figure 74: The track shapefile for clustering the KRDZ data.

KRDZGear.get_tracks_shp_reference

KRDZGear.**get_tracks_shp_reference**(*trk_shp*, *tile_poly*)

Get track shapefiles outside a buffer of a given tile (for point cloud data), used as a reference for clustering the KRDZ data within the tile.

Parameters

- **trk_shp** (*MultiLineString*) – A number of geometry objects of track lines.
- **tile_poly** (*Polygon*) – A tile.

Returns

Track lines data used as reference for clustering the KRDZ data.

Return type

tuple

Examples:

```
>>> from src.shaft import KRDZGear
>>> krdzg = KRDZGear()
>>> trk_shp_data = krdzg.trk.load_tracks_shp(elr=krdzg.elr)
>>> track_shp = MultiLineString(trk_shp_data.geometry.to_list())
>>> type(track_shp)
shapely.geometry.multilinestring.MultiLineString
>>> len(track_shp.geoms)
440
>>> tile_xy = (380600, 665400)
>>> tile_polygon = krdzg.get_pcd_tile_polygon(tile_xy=tile_xy)
>>> type(tile_polygon)
shapely.geometry.polygon.Polygon
>>> print(tile_polygon.bounds)
(380600.0, 665400.0, 380700.0, 665500.0)
>>> tile_tracks_shp, tile_tracks_shp_ref = krdzg.get_tracks_shp_reference(
...     trk_shp=track_shp, tile_poly=tile_polygon)
>>> type(tile_tracks_shp)
shapely.geometry.multilinestring.MultiLineString
>>> len(tile_tracks_shp.geoms)
2
>>> type(tile_tracks_shp_ref)
shapely.geometry.multilinestring.MultiLineString
>>> len(tile_tracks_shp_ref.geoms)
442
```

KRDZGear.import_classified_krdz

KRDZGear.**import_classified_krdz**(*update=False*, *verbose=True*, ***kwargs*)

Import the classified KRDZ rail head data into the project database.

Parameters

- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to `True`.
- **kwargs** – [Optional] parameters of pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.shaft import KRDZGear
>>> krdzg = KRDZGear()
>>> krdzg.import_classified_krdz(if_exists='replace', verbose=True)
To import classified rail head data into the table "PCD"."KRDZ_Classified"?
 [No]|Yes: yes
Importing the data ... Done.
```

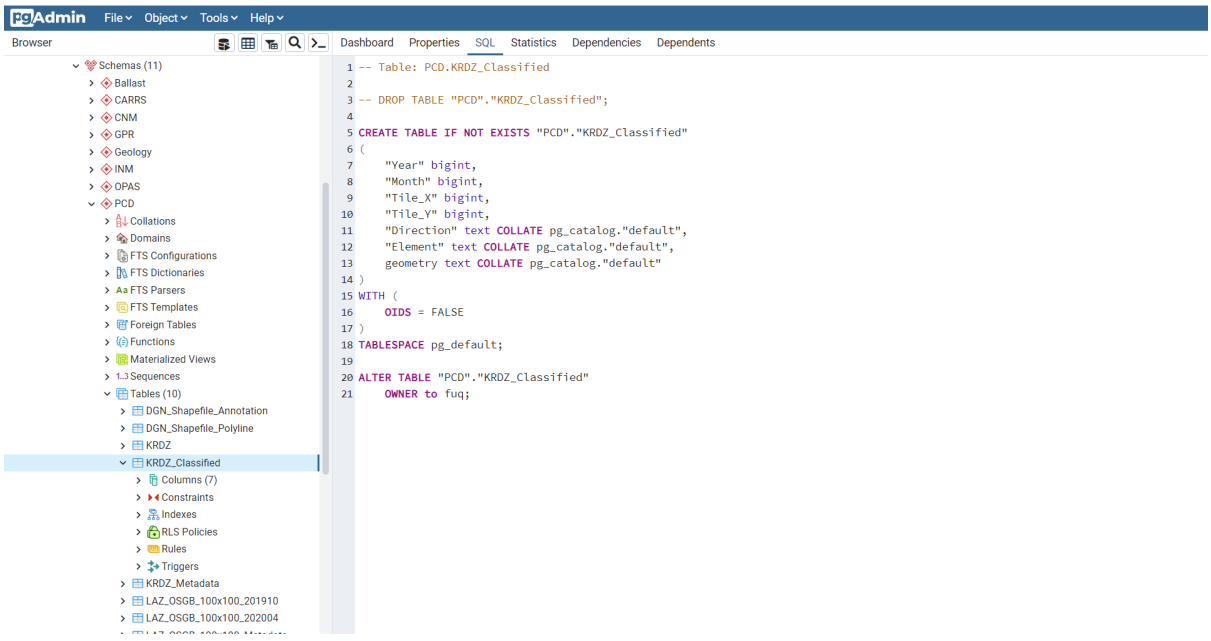


Figure 75: Snapshot of the "PCD"."KRDZ_Classified" table.

KRDZGear.load_classified_krdz

KRDZGear.**load_classified_krdz**(*tile_xy=None*, *pcd_date=None*, *direction=None*, *element=None*, *as_dict=False*)

Load data of classified KRDZ rail head data from the project database.

Parameters

- **tile_xy** (*tuple | list | str | None*) – Easting and northing of a tile for the point cloud data; defaults to `None`.
- **pcd_date** (*str | int | None*) – Date of the point cloud data; defaults to `None`.
- **direction** (*str | None*) – Railway direction; when `direction=None` (default), it refers to both up and down directions.
- **element** (*str | list | None*) – Element of rail head; when `element=None` (default), it refers to all available elements.
- **as_dict** (*bool*) – Whether to convert the retrieved dataframe to dict format; defaults to `False`

Returns

KRDZ rail head data (classified by up and down directions).

Return type

pandas.DataFrame

Examples:

```
>>> from src.shaft import KRDZGear
>>> krdzg = KRDZGear()
>>> krdz_dat_201910 = krdzg.load_classified_krdz(pcd_date='201910')
>>> krdz_dat_201910.head()
   Year  ...                                           geometry
0  2019  ...  LINESTRING Z (340199.193 674110.113 33.165, 34...
1  2019  ...  LINESTRING Z (340199.18 674110.147 33.15, 3401...
2  2019  ...  LINESTRING Z (340199.601 674111.877 33.163, 34...
3  2019  ...  LINESTRING Z (340199.613 674111.843 33.148, 34...
4  2019  ...  LINESTRING Z (340199.866 674111.17 33.163, 340...
[5 rows x 7 columns]
>>> krdz_dat_201910.shape
(9812, 7)
>>> krdz_x340500y674200_201910 = krdzg.load_classified_krdz(
...     tile_xy=(340500, 674200), pcd_date='201910')
>>> krdz_x340500y674200_201910.shape
(10, 7)
>>> krdz_x340500y674200_x364600y677500 = krdzg.load_classified_krdz(
...     tile_xy=[(340500, 674200), (364600, 677500)])
>>> krdz_x340500y674200_x364600y677500.shape
(40, 7)
>>> krdz_up = krdzg.load_classified_krdz(direction='up')
>>> krdz_up.shape
(9808, 7)
>>> krdz_up_top_202004 = krdzg.load_classified_krdz(
...     pcd_date='202004', direction='up', element=['left top', 'right top'])
>>> krdz_up_top_202004.shape
(1961, 7)
>>> krdz_up_top_202004.Element.unique().tolist()
['LeftTopOfRail', 'RightTopOfRail']
>>> krdz_dat = krdzg.load_classified_krdz()
>>> krdz_dat.shape
(19629, 7)
```

KRDZGear.load_pcd_krdz

KRDZGear.**load_pcd_krdz**(*tile_xy=None*, *pcd_date=None*)

Get (X, Y, Z) coordinates of the left and right tops, as well as the running edges, of the rail heads from the KRDZ data.

Parameters

- **tile_xy** (*tuple | list | str | None*) – X and Y coordinates in reference to a tile for the point cloud data; defaults to None.
- **pcd_date** (*str | int | None*) – Date of the point cloud data; defaults to None.

Returns

(X, Y, Z) coordinates of the left top, left running edge, right top, right running edge and center line.

Return type

dict

Examples:

```
>>> from src.shaft import KRDZGear
>>> krdzg = KRDZGear()
>>> krdz_xyz_dat = krdzg.load_pcd_krdz(pcd_date='202004')
>>> type(krdz_xyz_dat)
dict
>>> list(krdz_xyz_dat.keys())
['LeftTopOfRail',
 'LeftRunningEdge',
 'RightTopOfRail',
 'RightRunningEdge',
 'Centre']
>>> krdz_xyz_dat['LeftTopOfRail']
array([[340131.7340, 674088.6170, 33.3540],
       [340132.2030, 674088.7910, 33.3530],
       [340133.1410, 674089.1400, 33.3490],
       [340134.0780, 674089.4890, 33.3440],
       [340135.0150, 674089.8380, 33.3390],
       ...,
       [340136.4910, 674086.7520, 33.3330],
       [340135.5540, 674086.4030, 33.3350],
       [340134.6170, 674086.0530, 33.3370],
       [340133.6800, 674085.7040, 33.3400],
       [340132.7440, 674085.3540, 33.3440]])
>>> krdz_xyz_dat['LeftTopOfRail'].shape
(152628, 3)
>>> krdz_xyz_dat['LeftRunningEdge']
array([[340131.7220, 674088.6510, 33.3390],
       [340132.1910, 674088.8260, 33.3380],
       [340133.1280, 674089.1740, 33.3340],
       [340134.0650, 674089.5230, 33.3290],
       [340135.0020, 674089.8730, 33.3240],
       ...,
       [340136.4790, 674086.7860, 33.3180],
       [340135.5420, 674086.4370, 33.3200],
       [340134.6050, 674086.0870, 33.3220],
       [340133.6680, 674085.7380, 33.3250],
       [340132.7310, 674085.3890, 33.3290]])
>>> krdz_xyz_dat['LeftRunningEdge'].shape
(152628, 3)
>>> krdz_xyz_dat['RightTopOfRail']
array([[340131.1890, 674090.0240, 33.3550],
       [340131.6570, 674090.1990, 33.3540],
       [340132.5940, 674090.5470, 33.3500],
       [340133.5320, 674090.8950, 33.3460],
       [340134.4680, 674091.2440, 33.3420],
       ...,
       [340135.9440, 674088.1580, 33.3360],
       [340135.0070, 674087.8080, 33.3380],
       [340134.0700, 674087.4590, 33.3410],
       [340133.1330, 674087.1090, 33.3430],
       [340132.1960, 674086.7610, 33.3470]])
>>> krdz_xyz_dat['RightTopOfRail'].shape
(152628, 3)
>>> krdz_xyz_dat['RightRunningEdge']
array([[340131.2030, 674089.9900, 33.3390],
       [340131.6710, 674090.1650, 33.3380],
```

(continues on next page)

(continued from previous page)

```
       [340132.6080, 674090.5130, 33.3350],
       [340133.5450, 674090.8620, 33.3300],
       [340134.4820, 674091.2110, 33.3270],
       ...,
       [340135.9580, 674088.1250, 33.3210],
       [340135.0210, 674087.7750, 33.3230],
       [340134.0840, 674087.4260, 33.3250],
       [340133.1470, 674087.0760, 33.3280],
       [340132.2100, 674086.7270, 33.3320]])
>>> krdz_xyz_dat['RightRunningEdge'].shape
(152628, 3)
>>> krdz_xyz_dat['Centre']
array([[340131.4620, 674089.3210, 33.3550],
       [340131.9310, 674089.4950, 33.3530],
       [340132.8680, 674089.8440, 33.3500],
       [340133.8050, 674090.1920, 33.3450],
       [340134.7420, 674090.5420, 33.3410],
       ...,
       [340136.2180, 674087.4550, 33.3350],
       [340135.2810, 674087.1060, 33.3370],
       [340134.3440, 674086.7570, 33.3390],
       [340133.4070, 674086.4070, 33.3420],
       [340132.4700, 674086.0580, 33.3450]])
>>> krdz_xyz_dat['Centre'].shape
(152628, 3)
```

KRDZGear.view_classified_krdz

KRDZGear.**view_classified_krdz**(*tile_xy*, *pcd_date*, *projection='3d'*, *cmap_name='tab10'*, *add_title=False*, *save_as=None*, *dpi=600*, *verbose=False*, ***kwargs*)

View classified KRDZ data of the rail heads of point cloud data.

Parameters

- **tile_xy** (*tuple | list | str*) – X and Y coordinates in reference to a tile for the point cloud data.
- **pcd_date** (*str | int*) – Date of the point cloud data.
- **projection** (*str | None*) – Projection type of the subplot; defaults to `'3d'`.
- **cmap_name** (*str | None*) – Name of a matplotlib color map; defaults to `'tab10'`.
- **add_title** (*bool*) – Whether to add a title to the plot; defaults to `False`.
- **save_as** (*str | None*) – File format that the view is saved as; defaults to `None`.
- **dpi** (*int | None*) – DPI for saving image; defaults to `600`.

- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to `False`.
- **kwargs** – [Optional] additional parameters of the method `classify_krdz()`.

Examples:

```
>>> from src.shaft import KRDZGear
>>> from pyhelpers.settings import mpl_preferences
>>> mpl_preferences(backend='TkAgg')
>>> krdzg = KRDZGear()
>>> tile_xy = (340500, 674200)
>>> # 3D view
>>> # krdzg.view_classified_krdz(tile_xy, '201910', save_as=".svg",␣
↪verbose=True)
>>> krdzg.view_classified_krdz(tile_xy=tile_xy, pcd_date='201910')
```

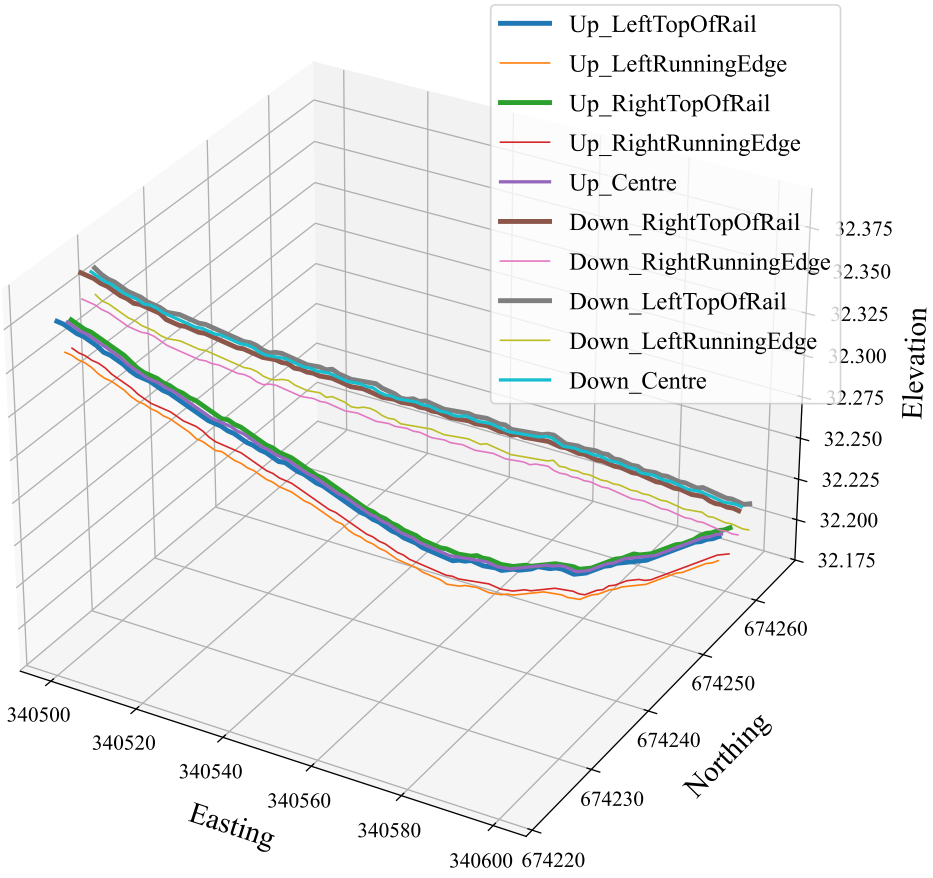


Figure 76: A 3D view of the classified KRDZ data in Tile (340500, 674200) in October 2019.

```
>>> # 2D plot - Vertical view
>>> # krdzg.view_classified_krdz(tile_xy, '201910', None, save_as=".svg",␣
↪verbose=True)
>>> krdzg.view_classified_krdz(tile_xy=tile_xy, pcd_date='201910',␣
↪projection=None)
```

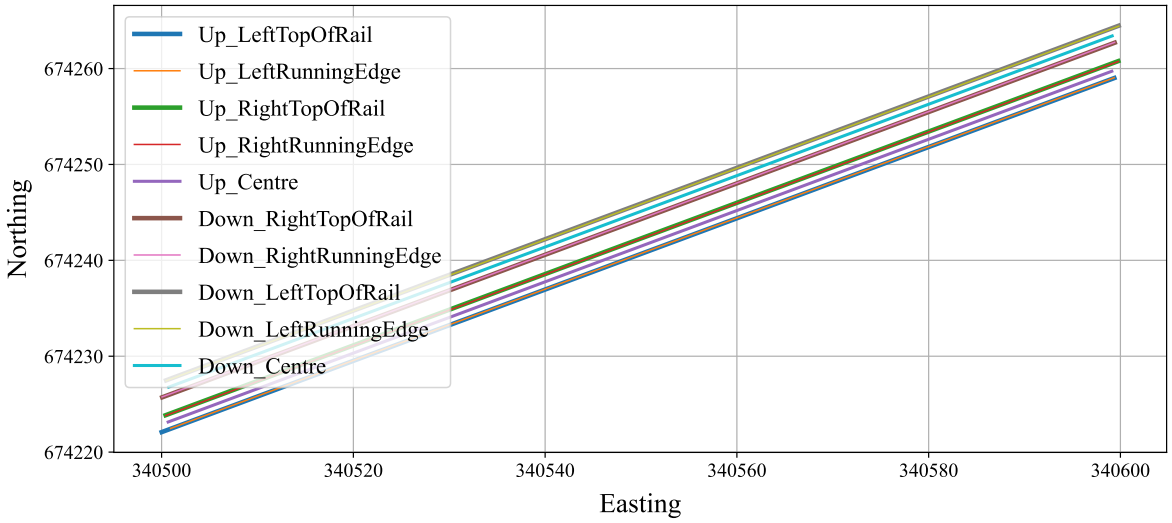


Figure 77: A vertical view of the classified KRDZ data in Tile (340500, 674200) in October 2019.

```
>>> # 3D view
>>> # krdzg.view_classified_krdz(tile_xy, '202004', save_as=".svg",␣
 ↪verbose=True)
>>> krdzg.view_classified_krdz(tile_xy=tile_xy, pcd_date='202004')
```

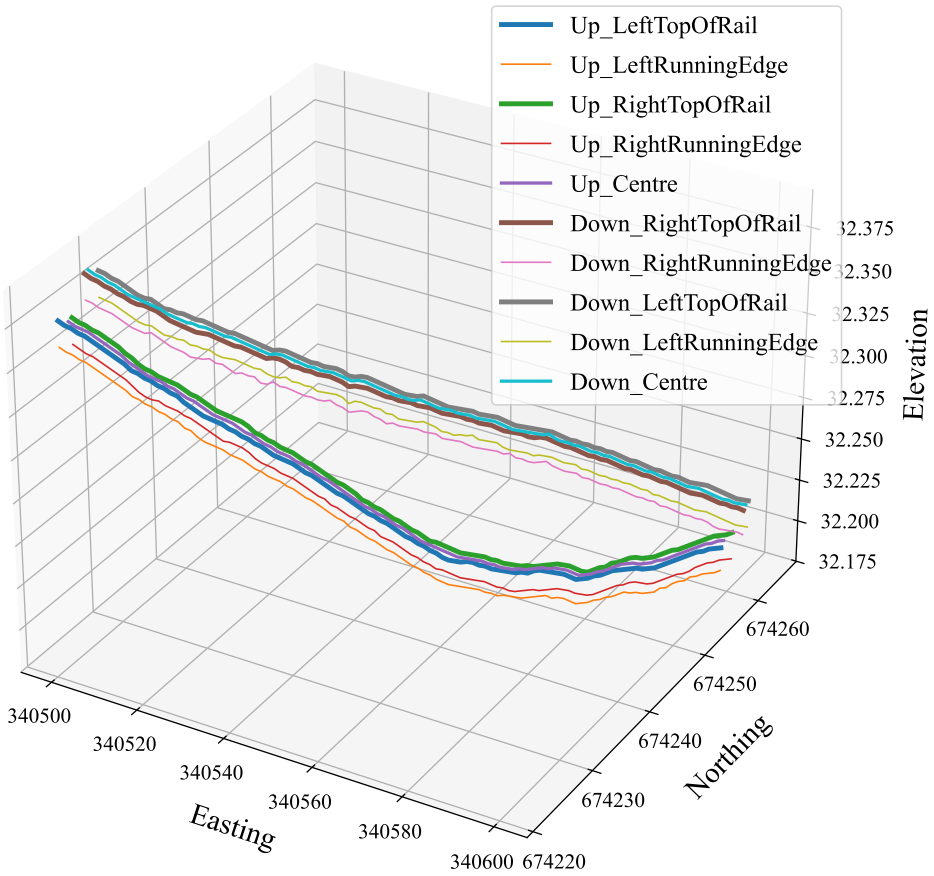


Figure 78: A 3D view of the classified KRDZ data in Tile (340500, 674200) in April 2020.

```
>>> # 2D plot - Vertical view
>>> # krdzg.view_classified_krdz(tile_xy, '202004', None, save_as=".svg",␣
 ↪verbose=True)
>>> krdzg.view_classified_krdz(tile_xy=tile_xy, pcd_date='202004',␣
 ↪projection=None)
```

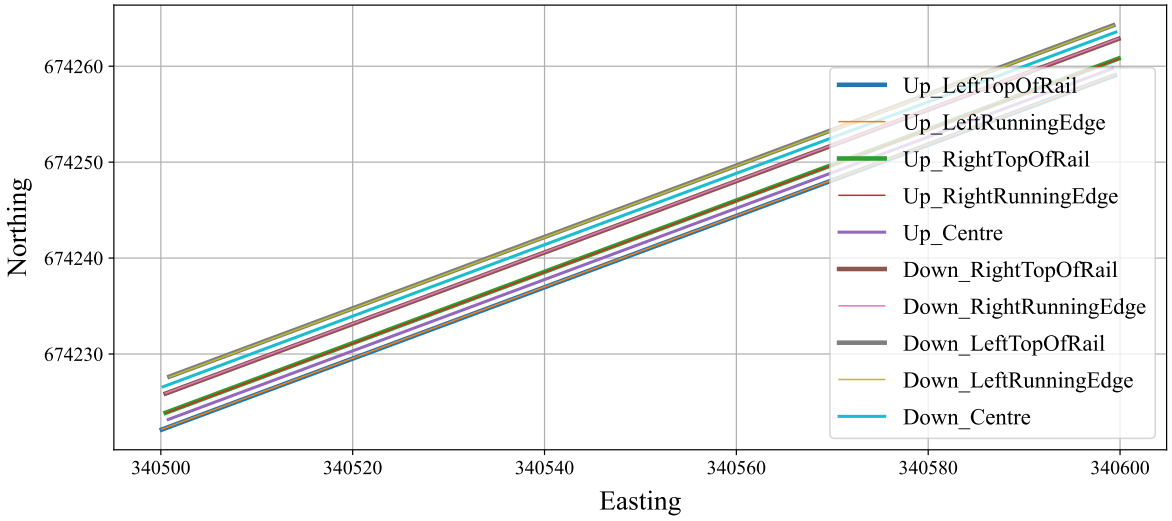


Figure 79: A vertical view of the classified KRDZ data in Tile (340500, 674200) in April 2020.

KRDZGear.view_pcd_krdz

KRDZGear.**view_pcd_krdz**(*tile_xy*, *pcd_date*, *projection='3d'*, *cmap_name='tab10'*, *add_title=False*, *save_as=None*, *dpi=600*, *verbose=False*, ***kwargs*)

Visualise original KRDZ data of the rail heads of point cloud data.

Parameters

- **tile_xy** (*tuple | list | str*) – X and Y coordinates in reference to a tile for the point cloud data.
- **pcd_date** (*str | int*) – Date of the point cloud data.
- **projection** (*str | None*) – Projection type of the subplot; defaults to `'3d'`.
- **cmap_name** (*str | None*) – Name of a matplotlib color map; defaults to `'tab10'`.
- **add_title** (*bool*) – Whether to add a title to the plot; defaults to `False`.
- **save_as** (*str | list | None*) – File format that the view is saved as; defaults to `None`.
- **dpi** (*int | None*) – DPI for saving image; defaults to `600`.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to `False`.
- **kwargs** – [Optional] additional parameters for the function matplotlib.pyplot.scatter.

Examples:

```
>>> from src.shaft import KRDZGear
>>> from pyhelpers.settings import mpl_preferences
>>> mpl_preferences(backend='TkAgg')
>>> krdzg = KRDZGear()
>>> tile_xy = (340500, 674200)
```

October 2019:

```
>>> # 3D view
>>> # krdzg.view_pcd_krdz(tile_xy, '201910', s=5, save_as=".svg", verbose=True)
>>> krdzg.view_pcd_krdz(tile_xy, '201910', projection='3d', s=5)
```

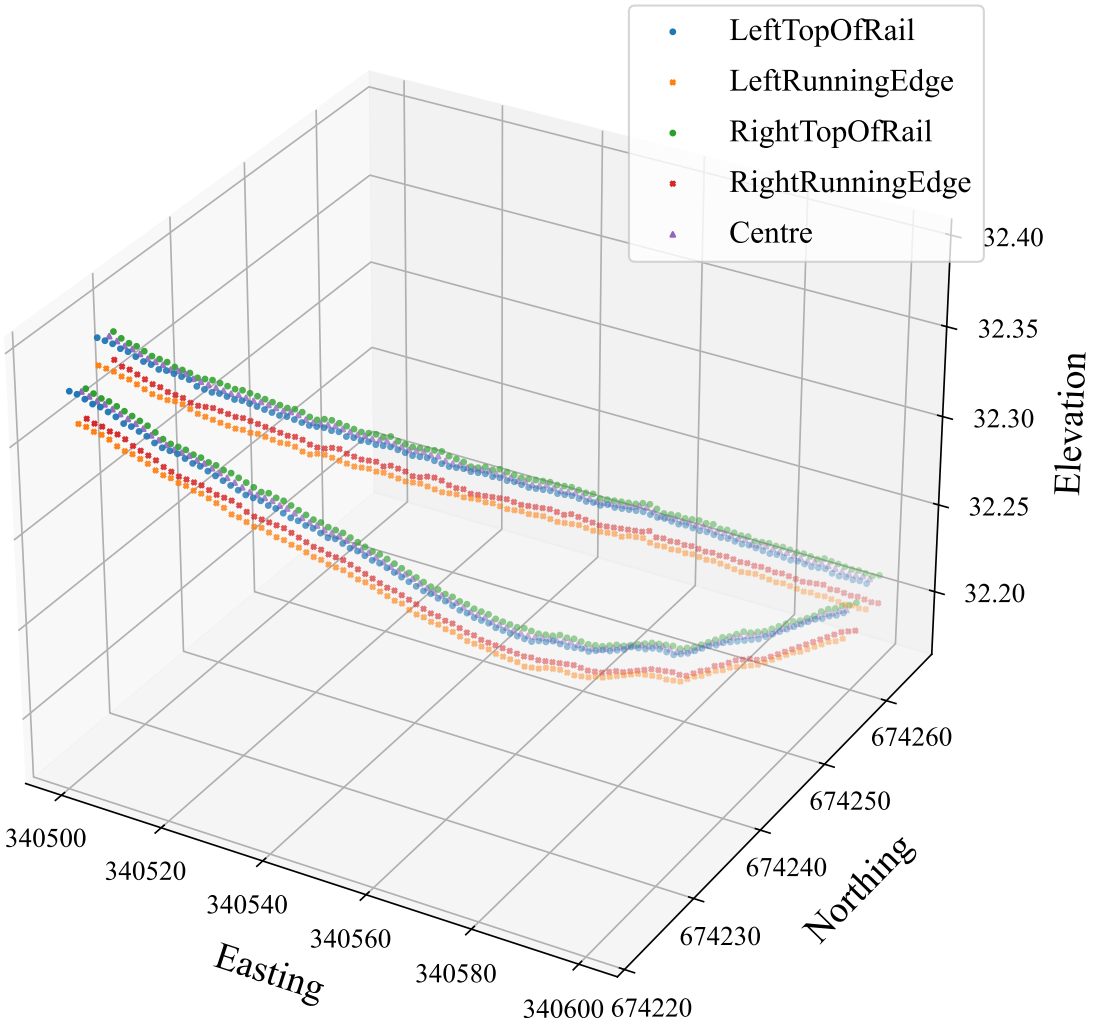


Figure 80: Original KRDZ data (3D) of Tile (340500, 674200) in October 2019.

```
>>> # 2D plot - Vertical view
>>> # krdzg.view_pcd_krdz(tile_xy, '201910', None, s=5, save_as=".svg",␣
 ↪verbose=True)
>>> krdzg.view_pcd_krdz(tile_xy, pcd_date='201910', projection=None, s=5)
```

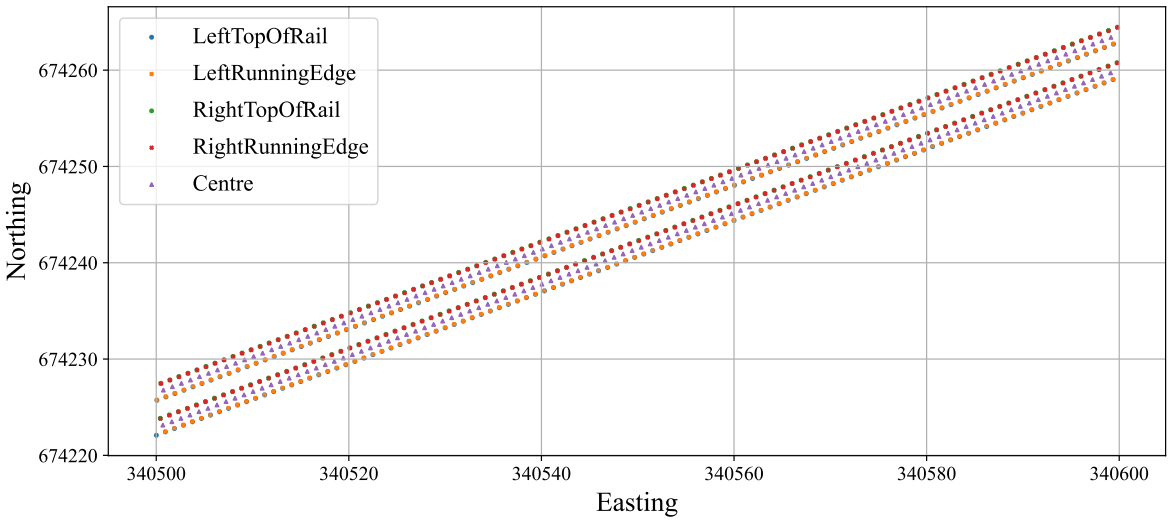


Figure 81: Original KRDZ data of Tile (340500, 674200) in October 2019.

April 2020:

```
>>> # 3D view
>>> # krdzg.view_pcd_krdz(tile_xy, '202004', s=5, save_as=".svg", verbose=True)
>>> krdzg.view_pcd_krdz(tile_xy, pcd_date='202004', projection='3d', s=5)
```

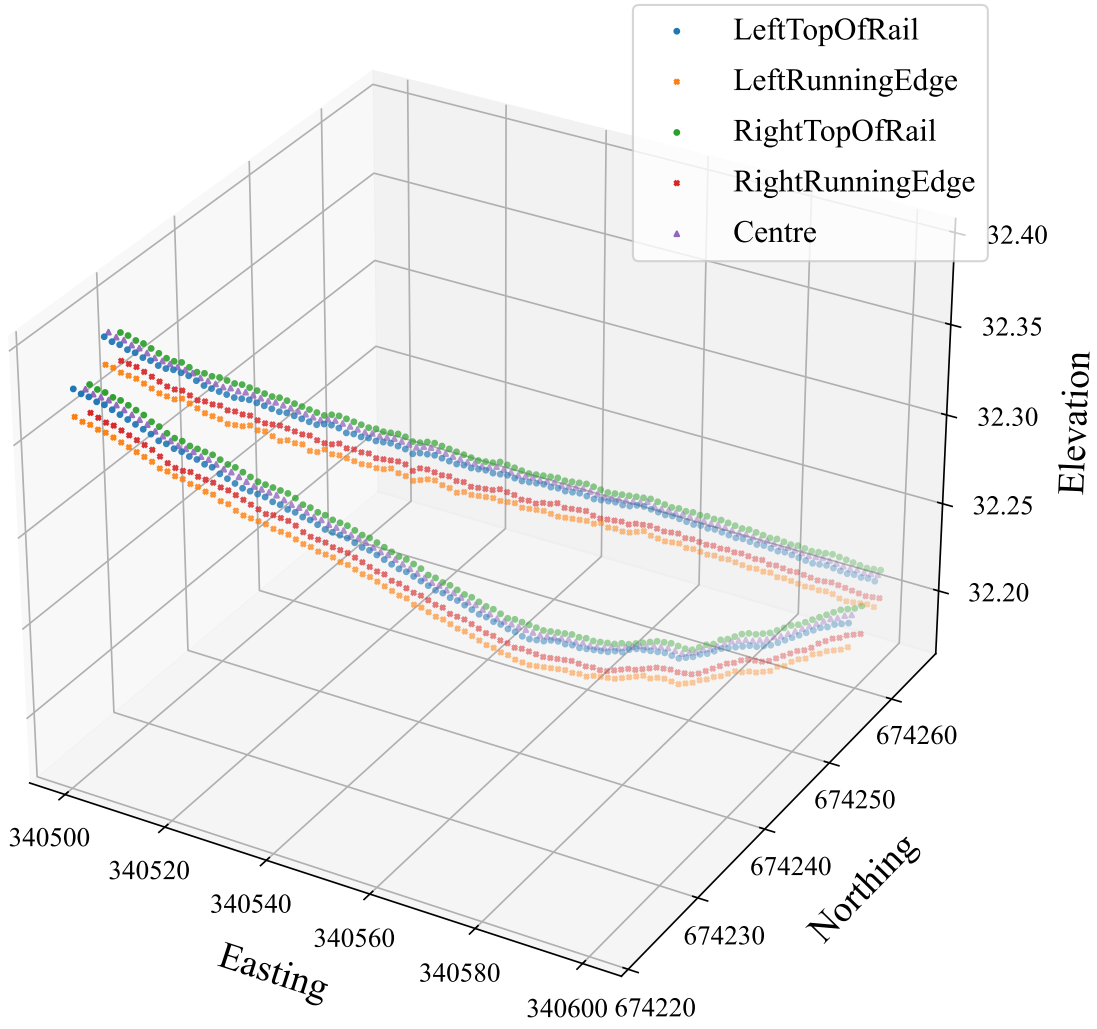


Figure 82: Original KRDZ data (3D) of Tile (340500, 674200) in April 2020.

```
>>> # 2D plot - Vertical view
>>> # krdzg.view_pcd_krdz(tile_xy, '202004', None, s=5, save_as=".svg",␣
 ↪verbose=True)
>>> krdzg.view_pcd_krdz(tile_xy, pcd_date='202004', projection=None, s=5)
```

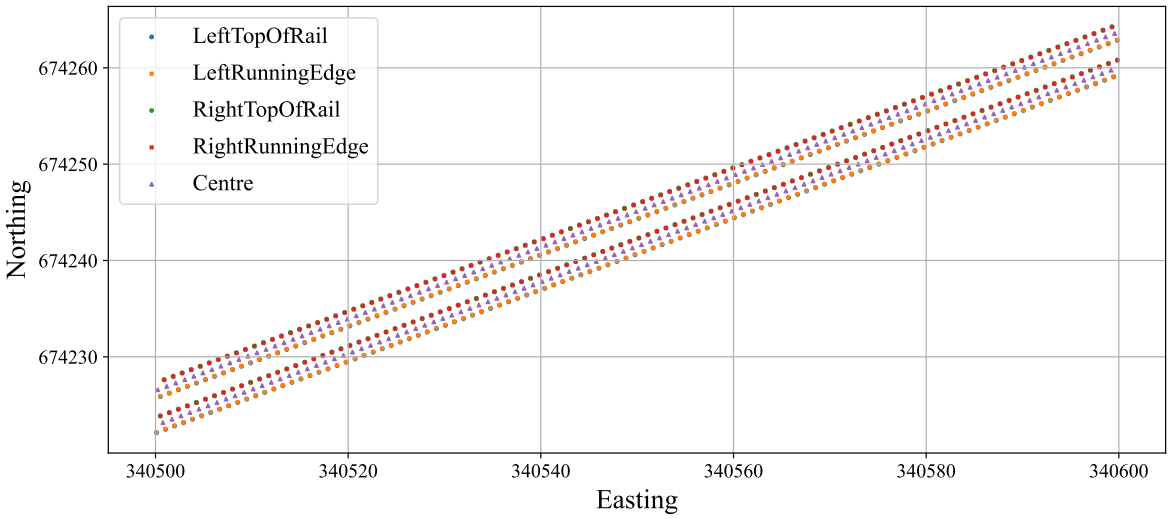


Figure 83: Original KRDZ data of Tile (340500, 674200) in April 2020.

TrackMovement

class src.shaft.**TrackMovement**(*elr='ECM8'*, *db_instance=None*)

Calculate track movement - the target to be predicted.

With the preprocessed point cloud data and relying on the classes *PCDHandler* and *KRDZGear*, this class leverages the KRDZ data to calculate track movement in both horizontal and vertical directions.

Parameters

- **elr** (*str*) – Engineer's Line Reference; defaults to `'ECM8'`.
- **db_instance** (TrackFixityDB | *None*) – PostgreSQL database instance; defaults to `None`.

Variables

elr (*str*) – Engineer's Line Reference.

Examples:

```
>>> from src.shaft import TrackMovement
>>> tm = TrackMovement()
>>> tm.NAME
'Calculation of track movement'
```

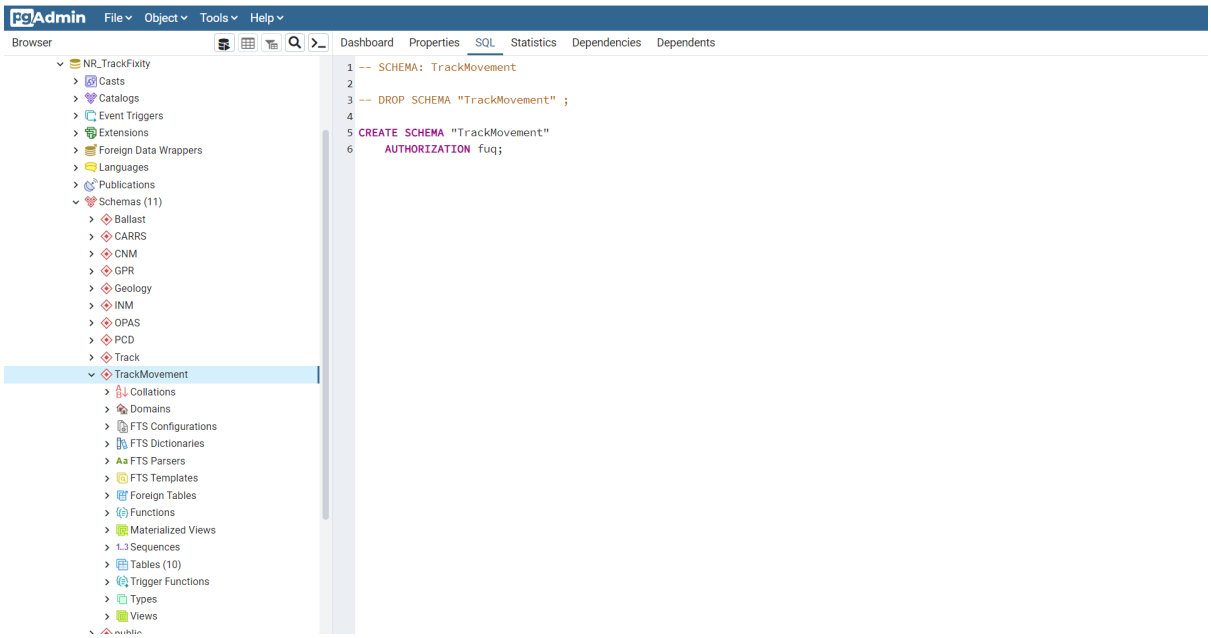


Figure 84: Snapshot of the *TrackMovement* schema.

Attributes:

NAME	Descriptive name of the class.
SCHEMA_NAME	Name of the schema for storing the calculated track movement data.

TrackMovement.NAME

TrackMovement.**NAME: str = 'Calculator for the track movement'**

Descriptive name of the class.

TrackMovement.SCHEMA_NAME

TrackMovement.**SCHEMA_NAME: str = 'TrackMovement'**

Name of the schema for storing the calculated track movement data.

Methods:

calculate_movement(element, direction[, ...])	Calculate average displacement about the movement of a given track section.
calculate_section_movement(section_original, ...)	Calculate average displacement about the movement of a track section for every subsection of a given length.
calculate_unit_subsection_displacement(...)	Calculate the displacement of a (short) section (of approx.
get_valid_table_names([element, direction, ...])	Get valid name(s) of the table(s) for storing track movement data of unit sections.
illustrate_unit_displacement(unit_section, ...)	Illustrate the lateral and vertical displacements of a (small) track section.
import_unit_movement([element, direction, ...])	Import the data of track movement of unit sections into the project database.
load_movement([element, direction, ...])	Load (and calculate) average displacement about the movement of a given track section.
split_section(section[, unit_length, ...])	Split a track section into a number of subsections, with each having approximately equal lengths.
view_heatmap(element, direction[, ...])	Create a heat map view of the track movement.
view_movement_violin_plot(data[, fig_size, ...])	Create a violin plot of the track movement.
weld_classified_krdz(element, direction, ...)	Weld the classified KRDZ rail head data for all tiles.

TrackMovement.calculate_movement

TrackMovement.**calculate_movement**(*element*, *direction*, *tile_xy=None*, *pcd_dates=None*, *ref_obj=None*, *len_offset=True*, *unit_length=1*, *subsect_len=10*, *rolling=False*, ***kwargs*)

Calculate average displacement about the movement of a given track section.

Parameters

- **element** (*str*) – Element of rail head, such as left or right top of rail or running edge.

- **direction** (*str*) – Railway direction, such as up or down direction.
- **tile_xy** (*tuple | list | str*) – Easting (X) and northing (Y) of the geographic Cartesian coordinates for a tile; defaults to `None`.
- **pcd_dates** (*list | tuple | None*) – Dates of the point cloud data to compare; defaults to `None`.
- **ref_obj** (*numpy.ndarray | LineString*) – Reference object for identifying direction of the displacement.
- **len_offset** (*bool*) – Whether to balance the section lengths; mostly, offset the data of longer length by the shorter one; defaults to `True`.
- **unit_length** (*int | float*) – Length (in metre) of each subsection; defaults to `1`.
- **subsect_len** (*int*) – Length (in metre) of a subsection for which movement is calculated; defaults to `10`.
- **rolling** (*bool*) – Whether to calculate the statistics on a rolling basis; defaults to `False`.
- **kwargs** (*bool*) – [Optional] parameters of the method `calculate_section_movement()`.

Returns

Average displacement about the movement of a given track section.

Return type

pandas.DataFrame

Examples:

```
>>> from src.shaft import TrackMovement
>>> tm = TrackMovement()
>>> ult_movements = tm.calculate_movement(
...     element='Left Top', direction='Up', multi_processes=True)
>>> ult_movements
      pseudo_id  ... vertical_displacement_cen_abs_max
0             0  ...                          -0.003203
1             1  ...                          -0.002798
2             2  ...                          -0.002200
3             3  ...                          -0.001305
4             4  ...                          -0.001000
...         ...  ...                                ...
7624       7624  ...                          -0.001239
7625       7625  ...                          -0.000825
7626       7626  ...                          -0.000912
7627       7627  ...                           0.000766
7628       7628  ...                           0.001267
[7629 rows x 38 columns]
>>> ult_movements_co = tm.calculate_movement(
...     element='Left Top', direction='Up', coarsely=True, multi_processes=True)
>>> ult_movements_co
```

(continues on next page)

(continued from previous page)

```
      pseudo_id  ... vertical_displacement_cen_abs_max
0             0  ...                          -0.003203
1             1  ...                          -0.002798
2             2  ...                          -0.002200
3             3  ...                          -0.001305
4             4  ...                          -0.001000
...         ...  ...                                ...
7626       7626  ...                          -0.001239
7627       7627  ...                          -0.000912
7628       7628  ...                          -0.000735
7629       7629  ...                           0.001267
7630       7630  ...                           0.001178
[7631 rows x 38 columns]
```

TrackMovement.calculate_section_movement

TrackMovement.**calculate_section_movement**(*section_original*, *section_shifted*, *ref_obj*, *unit_length=1*, *subsect_len=10*, *len_offset=True*, *rolling=False*, *multi_processes=False*, ***kwargs*)

Calculate average displacement about the movement of a track section for every subsection of a given length.

Parameters

- **section_original** (*numpy.ndarray | LineString*) – A track section's original position of a track at an earlier time.
- **section_shifted** (*numpy.ndarray | LineString*) – (Almost) the same track section's position of at a later time.
- **ref_obj** (*numpy.ndarray | LineString*) – A reference object for identifying direction of the displacement.
- **len_offset** (*bool*) – Whether to balance the section lengths; mostly, offset the data of longer length by the shorter one; defaults to `True`.
- **unit_length** (*int | float*) – Length (in metre) of each subsection; defaults to `1`.
- **subsect_len** (*int*) – Length (in metre) of a subsection for which movement is calculated; defaults to `10`.
- **rolling** (*bool*) – Whether to calculate the statistics on a rolling basis; defaults to `False`.
- **multi_processes** (*bool*) – Whether to use multiple CPUs (see also multiprocessing.Pool()); defaults to `False`.
- **kwargs** (*bool*) – [Optional] parameters of the method `split_section()`.

Returns
Basic statistics of track movement.

Return type
pandas.DataFrame

Examples:

```
>>> from src.shaft import TrackMovement
>>> from src.shaft.sec_utils import rearrange_line_points, length_offset
>>> from src.utils import geom_distance
>>> from pyhelpers.settings import mpl_preferences
>>> import functools
>>> mpl_preferences(backend='TkAgg')
>>> tm = TrackMovement()
>>> # KRDZ data of the rail heads in the down direction within the Tile (357500,
 ↪ 677700)
>>> krdz_x357500y677700_d = tm.load_classified_krdz((357500, 677700), direction=
 ↪'down')
>>> krdz_x357500y677700_d.head()
   Year  ...                                           geometry
0  2019  ...  LINESTRING Z (357599.306 677705.436 27.895, 35...
1  2019  ...  LINESTRING Z (357599.319 677705.469 27.875, 35...
2  2019  ...  LINESTRING Z (357599.867 677706.834 27.743, 35...
3  2019  ...  LINESTRING Z (357599.853 677706.8 27.731, 3575...
4  2019  ...  LINESTRING Z (357599.587 677706.136 27.819, 35...
[5 rows x 7 columns]
>>> # Top of the right rail in the down direction within (357500, 677700)
>>> expr = 'Element=="RightTopOfRail"'
>>> drt_x357500y677700 = krdz_x357500y677700_d.query(expr).set_index(['Year',
 ↪'Month'])
>>> # Data of Oct. 2019
>>> drt201910_ = drt_x357500y677700.loc[(2019, 10), 'geometry']
>>> # Data of Apr. 2020
>>> drt202004_ = drt_x357500y677700.loc[(2020, 4), 'geometry']
>>> # Adjust the order of points in the linestring
>>> drt201910, drt202004 = rearrange_line_points(drt201910_, drt202004_)
>>> drt201910.length
108.06188326010438
>>> drt202004.length
109.06372793933721
>>> # Get a reference object (for identifying the direction of displacement)
>>> ref_expr = 'Year==2019 and Element=="Centre"'
>>> ref_objects = krdz_x357500y677700_d.query(ref_expr).geometry.values[0]
>>> print(ref_objects.wkt)
LINESTRING Z (357599.587 677706.1360000001 27.819, 357598.656 677706.501 27.814,
 ↪ 35...
>>> # Illustrate an example 1-metre subsection
>>> drt201910_subs, drt202004_subs = map(
...     functools.partial(tm.split_section, use_original_points=False),
...     length_offset(drt201910, drt202004))
>>> drt201910_1m = drt201910_subs.geoms[15]
>>> drt202004_1m = min(drt202004_subs.geoms, key=geom_distance(drt201910_1m.
 ↪centroid))
>>> annot = 'top of the right rail'
>>> tm.illustrate_unit_displacement(drt201910_1m, drt202004_1m, element_
 ↪label=annot)
>>> # from pyhelpers.store import save_fig
```

(continues on next page)

(continued from previous page)

```
>>> # fig_pathname = "docs/source/_images/tm_calc_sect_movement_x357500y677700_
 ↪drt"
>>> # save_fig(f"{fig_pathname}.svg", transparent=True, verbose=True)
>>> # save_fig(f"{fig_pathname}.pdf", transparent=True, verbose=True)
```

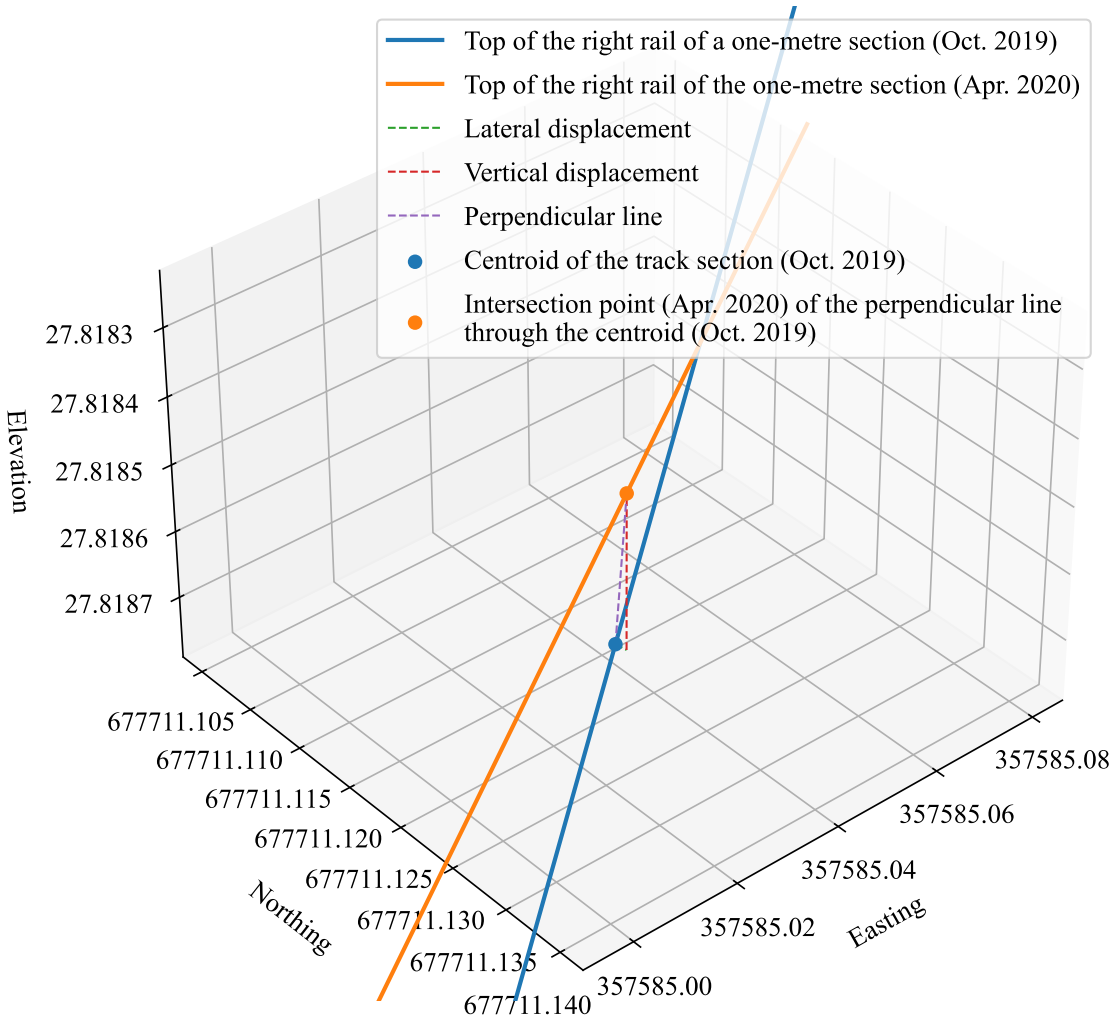


Figure 85: An exmaple of the lateral and vertical displacements of the top of right rail of a ~one-metre subsection in the down direction within the Tile (357500, 677700).

```
>>> # Average displacements for every 10-metre subsection
>>> drt_movement_x357500y677700 = tm.calculate_section_movement(
...     section_original=drt201910, section_shifted=drt202004, ref_obj=ref_
 ↪objects,
...     subsect_len=10)
>>> drt_movement_x357500y677700.head()
   pseudo_id  ... vertical_displacement_cen_abs_max
0          0  ...                          -0.001012
1          1  ...                          -0.000910
2          2  ...                          -0.001012
3          3  ...                          -0.001305
```

(continues on next page)

(continued from previous page)

```
4          4  ...                          -0.001405
[5 rows x 38 columns]
>>> drt_movement_x357500y677700.shape
(11, 38)
>>> # Rolling average of every 10 metres
>>> drt_movement_x357500y677700_ra = tm.calculate_section_movement(
...     section_original=drt201910, section_shifted=drt202004, ref_obj=ref_
 ↪objects,
...     subsect_len=10, rolling=True)
>>> drt_movement_x357500y677700_ra.shape
(99, 38)
```

TrackMovement.calculate_unit_subsection_displacement

static TrackMovement.**calculate_unit_subsection_displacement**(*unit_sec_orig*, *unit_sec_shifted*, *ref_obj*)

Calculate the displacement of a (short) section (of approx. one metre) between two different times.

Parameters

- **unit_sec_orig** (*numpy.ndarray | LineString*) – A unit track section's position at an earlier time.
- **unit_sec_shifted** (*numpy.ndarray | LineString*) – (Almost) the same unit track section' position at a later time.
- **ref_obj** (*numpy.ndarray | LineString*) – A reference object for identifying direction of the displacement.

Returns

Both lateral and vertical displacements.

Return type

tuple

Examples:

```
>>> from src.shaft import TrackMovement
>>> from src.shaft.sec_utils import rearrange_line_points
>>> from src.utils import geom_distance
>>> from pyhelpers.settings import mpl_preferences
>>> mpl_preferences(backend='TkAgg')
>>> tm = TrackMovement()
>>> # KRDZ data within the Tile (357500, 677700)
>>> krdz_x357500y677700 = tm.load_classified_krdz(tile_xy=(357500, 677700))
>>> krdz_x357500y677700.head()
   Year  ...                                           geometry
0  2019  ...  LINESTRING Z (357599.131 677701.85 28.137, 357...
1  2019  ...  LINESTRING Z (357599.144 677701.882 28.118, 35...
2  2019  ...  LINESTRING Z (357599.686 677703.249 27.988, 35...
3  2019  ...  LINESTRING Z (357599.672 677703.214 27.976, 35...
```

(continues on next page)

(continued from previous page)

```
4  2019  ...  LINESTRING Z (357599.409 677702.55 28.063, 357...
[5 rows x 7 columns]
>>> # Top of the left rail in the up direction within (357500, 677700)
>>> expr = 'Direction=="Up" and Element=="LeftTopOfRail"'
>>> ult_x357500y677700 = krdz_x357500y677700.query(expr).set_index(['Year',
 ↪'Month'])
>>> # Data of Oct. 2019
>>> ult201910 = ult_x357500y677700.loc[(2019, 10), 'geometry']
>>> # Data of Apr. 2020
>>> ult202004 = ult_x357500y677700.loc[(2020, 4), 'geometry']
>>> # Adjust the order of points in the linestring
>>> ult201910, ult202004 = rearrange_line_points(ult201910, ult202004)
>>> # Divide the linestring into one-metre subsections
>>> ult201910_subs = tm.split_section(section=ult201910, use_original_
 ↪points=False)
>>> len(ult201910_subs.geoms)
109
>>> ult202004_subs = tm.split_section(section=ult202004, use_original_
 ↪points=False)
>>> len(ult202004_subs.geoms)
109
>>> # An example of a one-metre subsection of the top of left rail in the up␣
 ↪direction
>>> ult201910_1m = ult201910_subs.geoms[50]
>>> ult201910_1m.length
0.9914271527767892
>>> print(ult201910_1m.wkt)
LINESTRING Z (357553.2933530097 677720.7146399085 27.88829200114355, 357552.
 ↪3846181...
>>> # The one-metre subsection in Apr. 2020 corresponding to the example in Oct.
 ↪ 2019
>>> ult202004_1m = min(ult202004_subs.geoms, key=geom_distance(ult201910_1m.
 ↪centroid))
>>> ult202004_1m.length
0.9914087119426246
>>> print(ult202004_1m.wkt)
LINESTRING Z (357553.3663216831 677720.6891952073 27.88775019639971, 357552.
 ↪4577717...
>>> # Illustrate the distance between the two one-metre data
>>> annot = 'top of left rail'
>>> tm.illustrate_unit_displacement(ult201910_1m, ult202004_1m, element_
 ↪label=annot)
>>> # from pyhelpers.store import save_fig
>>> # fig_pathname = "docs/source/_images/tm_calc_unit_subsect_disp_
 ↪x357500y677700_demo"
>>> # save_fig(f"{fig_pathname}.svg", transparent=True, verbose=True)
>>> # save_fig(f"{fig_pathname}.pdf", transparent=True, verbose=True)
```

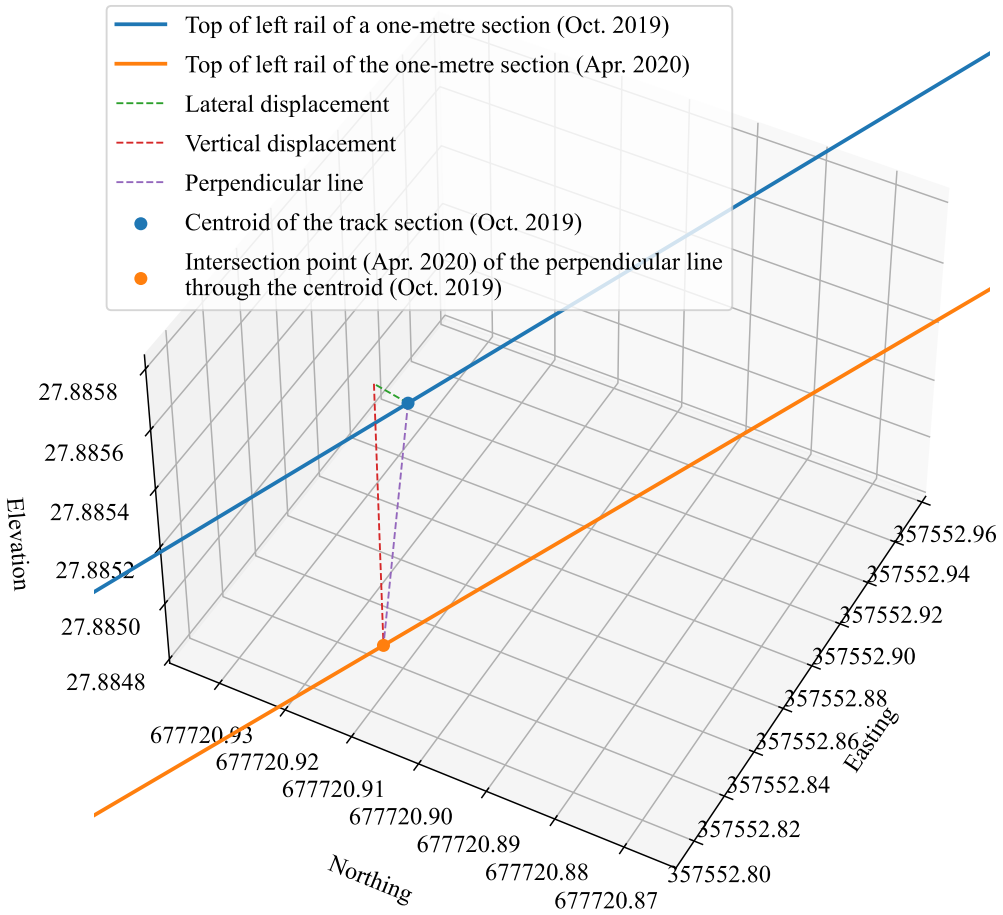


Figure 86: Movement of the top of the left rail of a ~one-metre track section in the up direction within the Tile (357500, 677700).

```
>>> # Get a reference object (for identifying the direction of displacement)
>>> ref_expr = 'Year==2019 and Direction=="Up" and Element=="Centre"'
>>> ref_objects = krdz_x357500y677700.query(ref_expr)['geometry'].iloc[0]
>>> # Lateral and vertical displacements
>>> displacement_data = tm.calculate_unit_subsection_displacement(
...     ult201910_1m, ult202004_1m, ref_objects)
>>> lateral_disp, vertical_disp, abs_min_disp = displacement_data
>>> # Lateral displacements of one end, centroid and the other end of the 1-m␣
 ↪section
>>> lateral_disp
[-0.006304874255657488, -0.005884731608958095, -0.0060948028789788]
>>> # Vertical displacements of one end, centroid and the other end of the 1-m␣
 ↪section
>>> vertical_disp
[-0.0008873096940130941, -0.0009624371630553282, -0.0009248734283608409]
```

TrackMovement.get_valid_table_names

TrackMovement.**get_valid_table_names**(*element=None*, *direction=None*, *pcd_dates=None*, *ret_input=False*)

Get valid name(s) of the table(s) for storing track movement data of unit sections.

Parameters

- **element** (*str | list | None*) – Element of rail head, e.g. left/right top of rail or running edge; defaults to None.
- **direction** (*str | list | None*) – Railway direction, e.g. up and down directions; defaults to None.
- **pcd_dates** (*list | tuple | None*) – Dates of the point cloud data to compare; defaults to None.
- **ret_input** – Whether to return validated names of the input element and direction; defaults to False.

Returns

Valid name(s) of the table(s) (for storing track movement data of unit sections) and, when ret_input=True, validated names of element, direction and pcd_dates.

Return type

list | tuple

Examples:

```
>>> from src.shaft import TrackMovement
>>> tm = TrackMovement()
>>> valid_table_names = tm.get_valid_table_names()
>>> valid_table_names
['Up_LeftTopOfRail_201910_202004',
 'Up_LeftRunningEdge_201910_202004',
 'Up_RightTopOfRail_201910_202004',
 'Up_RightRunningEdge_201910_202004',
 'Up_Centre_201910_202004',
 'Down_LeftTopOfRail_201910_202004',
 'Down_LeftRunningEdge_201910_202004',
 'Down_RightTopOfRail_201910_202004',
 'Down_RightRunningEdge_201910_202004',
 'Down_Centre_201910_202004']
>>> valid_table_names = tm.get_valid_table_names(element=['left top', 'right top
↪'])
>>> valid_table_names
['Up_LeftTopOfRail_201910_202004',
 'Up_RightTopOfRail_201910_202004',
 'Down_LeftTopOfRail_201910_202004',
 'Down_RightTopOfRail_201910_202004']
```

TrackMovement.illustrate_unit_displacement

static TrackMovement.**illustrate_unit_displacement**(*unit_sec_orig*, *unit_sec_shifted*, *len_offset=False*, *element_label=None*, *add_title=False*, ***kwargs*)

Illustrate the lateral and vertical displacements of a (small) track section.

Parameters

- **unit_sec_orig** (*LineString*) – Track section (i.e. original position of a track at an earlier time).
- **unit_sec_shifted** (*LineString*) – Track section (i.e. position the track at a later time).
- **len_offset** (*bool*) – Whether to len_offset the data of longer length by the shorter one; defaults to `False`.
- **element_label** (*str | None*) – Label of the rail head's element to be illustrated; defaults to `None`.
- **add_title** (*bool*) – Whether to add a title to the plot; defaults to `False`.
- **kwargs** (*float | int | None*) – [Optional] additional parameters for the function `length_offset()`.

Examples:

```
>>> from src.shaft.movement import TrackMovement
>>> from src.shaft.sec_utils import length_offset, rearrange_line_points
>>> from shapely.geometry import LineString
>>> from pyhelpers.settings import mpl_preferences
>>> mpl_preferences(backend='TkAgg')
>>> tm = TrackMovement()
>>> # Left top of rail in the up direction within tile (340500, 674200)
>>> tile_xy = (340500, 674200)
>>> direction = 'Up'
>>> element = 'Left Top'
>>> krdz_ult_x340500y674200 = tm.load_classified_krdz(tile_xy, None, direction,␣
↪element)
>>> ult_201910_ = krdz_ult_x340500y674200.query('Year == 2019')['geometry'].
↪iloc[0]
>>> type(ult_201910_)
shapely.geometry.linestring.LineString
>>> ult_201910_.length
105.9959864289052
>>> ult_202004_ = krdz_ult_x340500y674200.query('Year == 2020')['geometry'].
↪iloc[0]
>>> type(ult_202004_)
shapely.geometry.linestring.LineString
>>> ult_202004_.length
105.99856695815154
>>> # Offset the section length and rearrange the lines
```

(continues on next page)

(continued from previous page)

```
>>> ult_201910, ult_202004 = length_offset(ult_201910_, ult_202004_)
>>> ult_201910, ult_202004 = rearrange_line_points(ult_201910, ult_202004)
>>> # In this example, the two sections remain as is due to little difference␣
 ↪in lengths
>>> ult_201910.length
105.9959864289052
>>> ult_202004.length
105.99856695815154
>>> # Divide into ~one-metre subsections
>>> ult_201910_subs, ult_202004_subs = map(tm.split_section, [ult_201910, ult_
 ↪202004])
>>> # An example 1m subsection of the top of left rail in the up direction in␣
 ↪Oct. 2019
>>> ult_201910_1m = ult_201910_subs.geoms[100]
>>> # A corresponding 1m subsection in Apr. 2020
>>> ult_202004_1m = ult_202004_subs.geoms[100]
>>> annot = 'top of the left rail'
>>> tm.illustrate_unit_displacement(ult_201910_1m, ult_202004_1m, element_
 ↪label=annot)
>>> # from pyhelpers.store import save_fig
>>> # fig_pathname = "docs/source/_images/tm_illust_unit_disp_x340500y674200_ult
 ↪"
>>> # save_fig(f"{fig_pathname}.svg", transparent=True, verbose=True)
>>> # save_fig(f"{fig_pathname}.pdf", transparent=True, verbose=True)
```

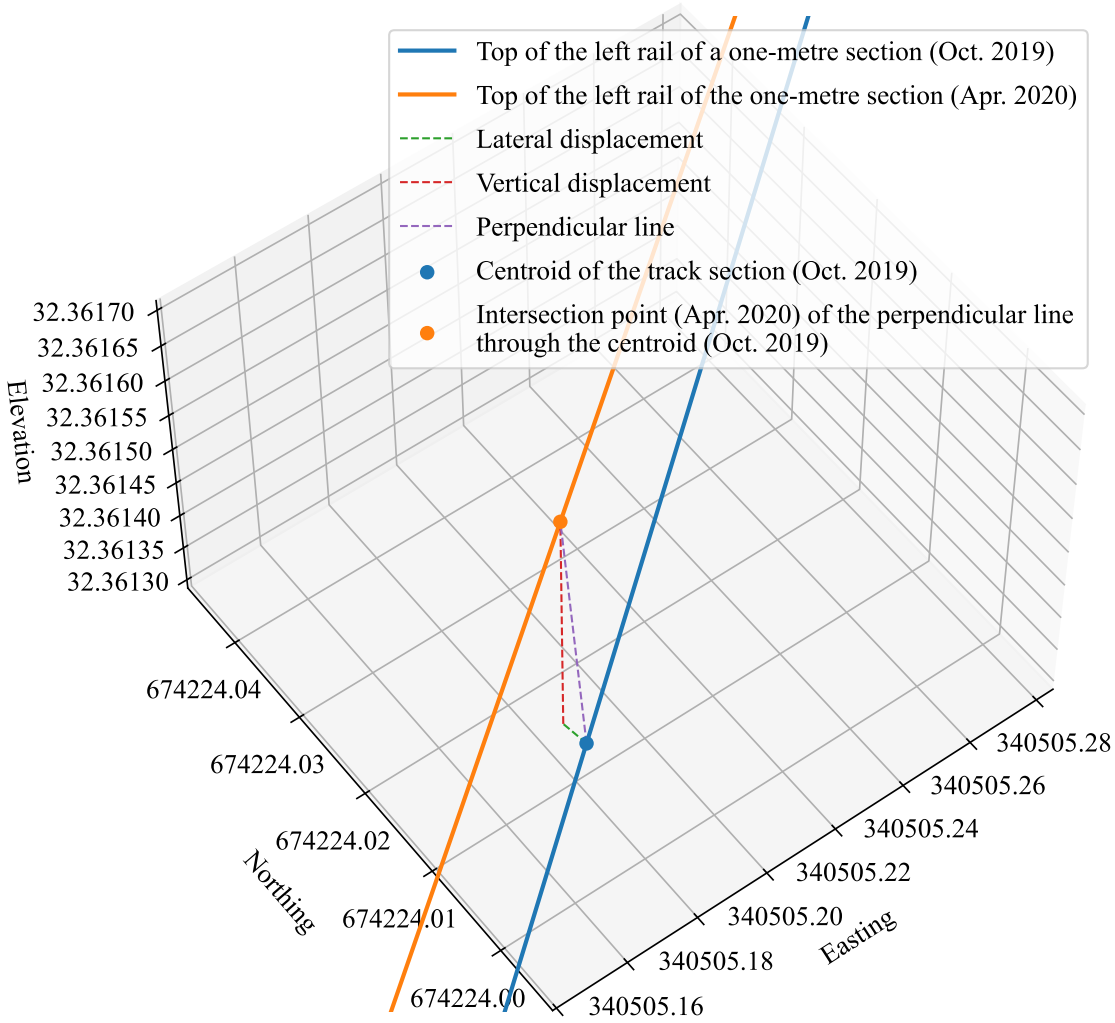


Figure 87: Movement of the top of the left rail of a ~one-metre track section in the up direction within the Tile (340500, 674200).

TrackMovement.import_unit_movement

`TrackMovement.`**`import_unit_movement`**(*element=None*, *direction=None*, *pcd_dates=None*, *confirmation_required=True*, *verbose=True*, *rolling=False*, ***kwargs*)

Import the data of track movement of unit sections into the project database.

Parameters

- **`element`** (*`str` | `None`*) – Element of rail head, e.g. left/right top of rail or running edge; defaults to `None`.
- **`direction`** (*`str` | `None`*) – Railway direction, e.g. up and down directions; defaults to `None`.
- **`pcd_dates`** (*`list` | `tuple` | `None`*) – Dates of the point cloud data to compare; defaults to `None`.

- **confirmation_required** (*bool*) – Whether asking for confirmation to proceed; defaults to `True`.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to `True`.
- **rolling** (*bool*) – Whether to calculate the statistics on a rolling basis; defaults to `False`.
- **kwargs** – [Optional] parameters of pyhelpers.dbms.PostgreSQL.import_data.

Examples:

```
>>> from src.shaft import TrackMovement
>>> tm = TrackMovement()
>>> tm.import_unit_movement(if_exists='replace')
To import unit movement data into the following tables in the schema
 ↪"TrackMovement":
    "Up_LeftTopOfRail_201910_202004"
    "Up_LeftRunningEdge_201910_202004"
    "Up_RightTopOfRail_201910_202004"
    "Up_RightRunningEdge_201910_202004"
    "Up_Centre_201910_202004"
    "Down_LeftTopOfRail_201910_202004"
    "Down_LeftRunningEdge_201910_202004"
    "Down_RightTopOfRail_201910_202004"
    "Down_RightRunningEdge_201910_202004"
    "Down_Centre_201910_202004"
? [No]|Yes: yes
Importing the data ...
    Up_LeftTopOfRail_201910_202004 ... Done.
    Up_LeftRunningEdge_201910_202004 ... Done.
    Up_RightTopOfRail_201910_202004 ... Done.
    Up_RightRunningEdge_201910_202004 ... Done.
    Up_Centre_201910_202004 ... Done.
    Down_LeftTopOfRail_201910_202004 ... Done.
    Down_LeftRunningEdge_201910_202004 ... Done.
    Down_RightTopOfRail_201910_202004 ... Done.
    Down_RightRunningEdge_201910_202004 ... Done.
    Down_Centre_201910_202004 ... Done.
```

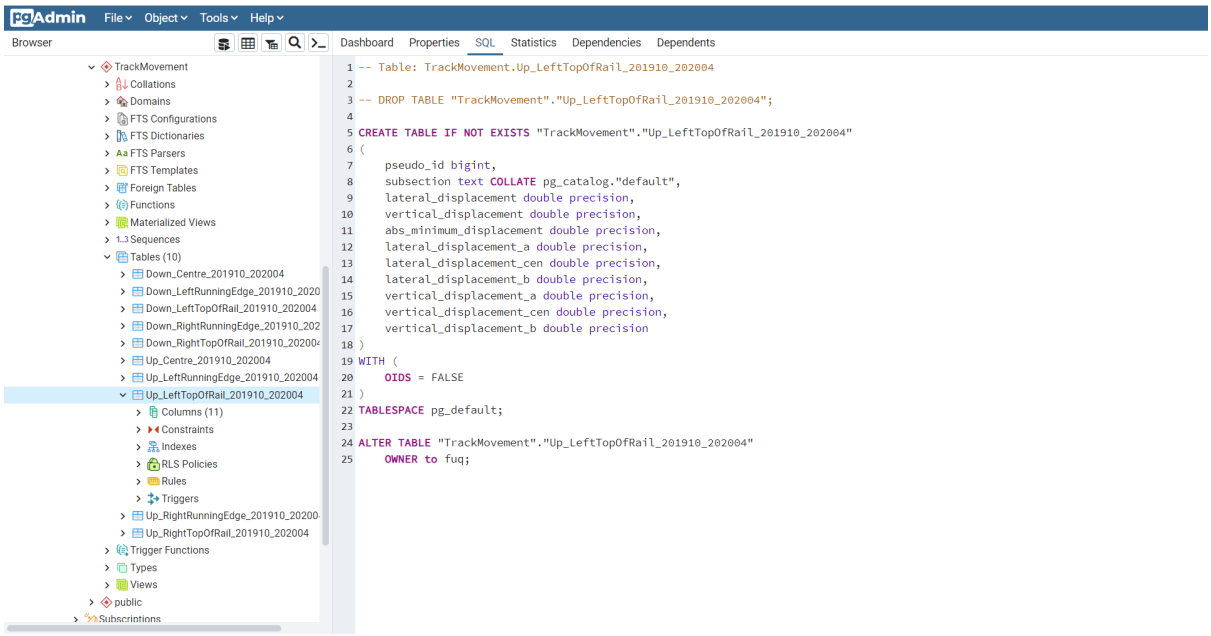


Figure 88: Snapshot of the "TrackMovement"."Up_LeftTopOfRail_201910_202004" table.

TrackMovement.load_movement

TrackMovement.**load_movement**(*element=None*, *direction=None*, *pcd_dates=None*, *subsect_len=10*, *rolling=False*, *keep_pseudo_id=False*, ***kwargs*)

Load (and calculate) average displacement about the movement of a given track section.

Parameters

- **element** (*str | list | None*) – Element of rail head, e.g. left/right top of rail or running edge; defaults to None.
- **direction** (*str | list | None*) – Railway direction, e.g. up and down directions; defaults to None.
- **pcd_dates** (*list | tuple | None*) – Dates of the point cloud data to compare; defaults to None.
- **subsect_len** (*int*) – Length (in metre) of a subsection for which movement is calculated; defaults to 10.
- **rolling** (*bool*) – Whether to calculate the statistics on a rolling basis; defaults to False.
- **keep_pseudo_id** (*bool*) – Whether to keep pseudo_id for unit subsections; defaults to False.
- **kwargs** – [Optional] parameters of pyhelpers.dbms.PostgreSQL.read_sql_query.

Returns
Average displacement about the movement of a given track section.

Return type
dict

Examples:

```
>>> from src.shaft import TrackMovement
>>> tm = TrackMovement()
>>> ult_movement = tm.load_movement(element='Left Top', direction='Up')
>>> type(ult_movement)
dict
>>> list(ult_movement.keys())
['Up_LeftTopOfRail_201910_202004']
>>> ult_movement['Up_LeftTopOfRail_201910_202004']
                                    subsection  ...  vertical_displacement_b_
↪abs_max
0     LINESTRING Z (399428.96 653473.9 34.442...  ...                        -0.
↪003203
1     LINESTRING Z (399434.5201166851 653482....  ...                        -0.
↪002794
2     LINESTRING Z (399440.0608295194 653490....  ...                        -0.
↪003202
3     LINESTRING Z (399445.5700898784 653498....  ...                        -0.
↪001803
4     LINESTRING Z (399451.0456262118 653507....  ...                        -0.
↪001201
                                              ...  ...                         ␣
↪    ...
7625  LINESTRING Z (340180.451306418 674103.1...  ...                        -0.
↪001000
7626  LINESTRING Z (340171.0809418422 674099....  ...                        -0.
↪001650
7627  LINESTRING Z (340161.7096778383 674096....  ...                        -0.
↪000914
7628  LINESTRING Z (340152.3387371677 674092....  ...                        -0.
↪000737
7629  LINESTRING Z (340142.9690238989 674089....  ...                         0.
↪001267
[7630 rows x 37 columns]
>>> ult_movement_ra = tm.load_movement(element='Left Top', direction='Up',␣
↪rolling=True)
>>> ult_movement_ra['Up_LeftTopOfRail_201910_202004_RA']
                                    subsection  ...  vertical_displacement_b_
↪abs_max
0      LINESTRING Z (399428.96 653473.9 34.44...  ...                        -0.
↪003203
1      LINESTRING Z (399429.5183848582 653474...  ...                        -0.
↪003203
2      LINESTRING Z (399430.0744669316 653475...  ...                        -0.
↪003203
3      LINESTRING Z (399430.6317003522 653476...  ...                        -0.
↪003001
4      LINESTRING Z (399431.1884705836 653477...  ...                        -0.
↪003001
                                              ...  ...                         ␣
↪    ...
```

(continues on next page)

(continued from previous page)

```
76286  LINESTRING Z (340146.7167791607 674090...  ...                          0.
↪001265
76287  LINESTRING Z (340145.7800034651 674090...  ...                          0.
↪001265
76288  LINESTRING Z (340144.8429013437 674089...  ...                          0.
↪001265
76289  LINESTRING Z (340143.9061259775 674089...  ...                          0.
↪001265
76290  LINESTRING Z (340142.9690238989 674089...  ...                          0.
↪001267
[76291 rows x 37 columns]
>>> trk_movement_100sl = tm.load_movement(subsect_len=100, keep_pseudo_id=True)
>>> len(trk_movement_100sl)
10
>>> list(trk_movement_100sl.keys())
['Up_LeftTopOfRail_201910_202004',
 'Up_LeftRunningEdge_201910_202004',
 'Up_RightTopOfRail_201910_202004',
 'Up_RightRunningEdge_201910_202004',
 'Up_Centre_201910_202004',
 'Down_LeftTopOfRail_201910_202004',
 'Down_LeftRunningEdge_201910_202004',
 'Down_RightTopOfRail_201910_202004',
 'Down_RightRunningEdge_201910_202004',
 'Down_Centre_201910_202004']
>>> trk_movement_100sl['Up_LeftTopOfRail_201910_202004']
                                      pseudo_id  ... vertical_displacement_b_
↪abs_max
0    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12...  ...                         -0.
↪003203
1    [100, 101, 102, 103, 104, 105, 106, 107, ...  ...                         -0.
↪005816
2    [200, 201, 202, 203, 204, 205, 206, 207, ...  ...                         -0.
↪003543
3    [300, 301, 302, 303, 304, 305, 306, 307, ...  ...                         -0.
↪002782
4    [400, 401, 402, 403, 404, 405, 406, 407, ...  ...                         -0.
↪004486
..                                             ...  ...                          ␣
↪   ...
758  [75800, 75801, 75802, 75803, 75804, 75805...  ...                         -0.
↪006993
759  [75900, 75901, 75902, 75903, 75904, 75905...  ...                         -0.
↪004793
760  [76000, 76001, 76002, 76003, 76004, 76005...  ...                         -0.
↪004802
761  [76100, 76101, 76102, 76103, 76104, 76105...  ...                         -0.
↪004801
762  [76200, 76201, 76202, 76203, 76204, 76205...  ...                         -0.
↪001740
[763 rows x 38 columns]
```

TrackMovement.split_section

static TrackMovement.**split_section**(*section*, *unit_length=1*, *coarsely=False*, *use_original_points=True*, *to_geoms=False*)

Split a track section into a number of subsections, with each having approximately equal lengths.

Parameters

- **section** (*numpy.ndarry | LineString*) – Track section (represented as a polyline).
- **unit_length** (*int | float*) – Length of each subsection; defaults to 1 (in metre).
- **coarsely** (*bool*) – Whether to split the section in a coarse way; defaults to False.
- **use_original_points** (*bool*) – Whether to use original points as splitters; defaults to False.
- **to_geoms** (*bool*) – Whether to transform the obtained geometry object (GeometryCollection) to its iterable form (GeometrySequence); defaults to True.

Returns

A sequence of approximately equal-length subsections of the polyline object.

Return type

list | GeometryCollection

Examples:

```
>>> from src.shaft import TrackMovement
>>> from src.shaft.sec_utils import weld_subsections
>>> tm = TrackMovement()
```

Example 1 - A single tile:

```
>>> tile_xy = (340500, 674200)
>>> # Top of the left rail in the up direction within tile (340500, 674200) in␣
↪10/2019
>>> krdz_ult_x340500y674200_201910 = tm.load_classified_krdz(
...     tile_xy=tile_xy, pcd_date='201910', direction='up', element='left top')
>>> ult_201910 = krdz_ult_x340500y674200_201910.loc[0, 'geometry']
>>> type(ult_201910)
shapely.geometry.linestring.LineString
>>> print("~%.4fm" % ult_201910.length)
~105.9960m
>>> # Get subsections, with each being approximately 10-metre long
>>> ult_201910_subs = tm.split_section(section=ult_201910, unit_length=10)
>>> type(ult_201910_subs)
shapely.geometry.collection.GeometryCollection
>>> len(ult_201910_subs.geoms)
```

(continues on next page)

(continued from previous page)

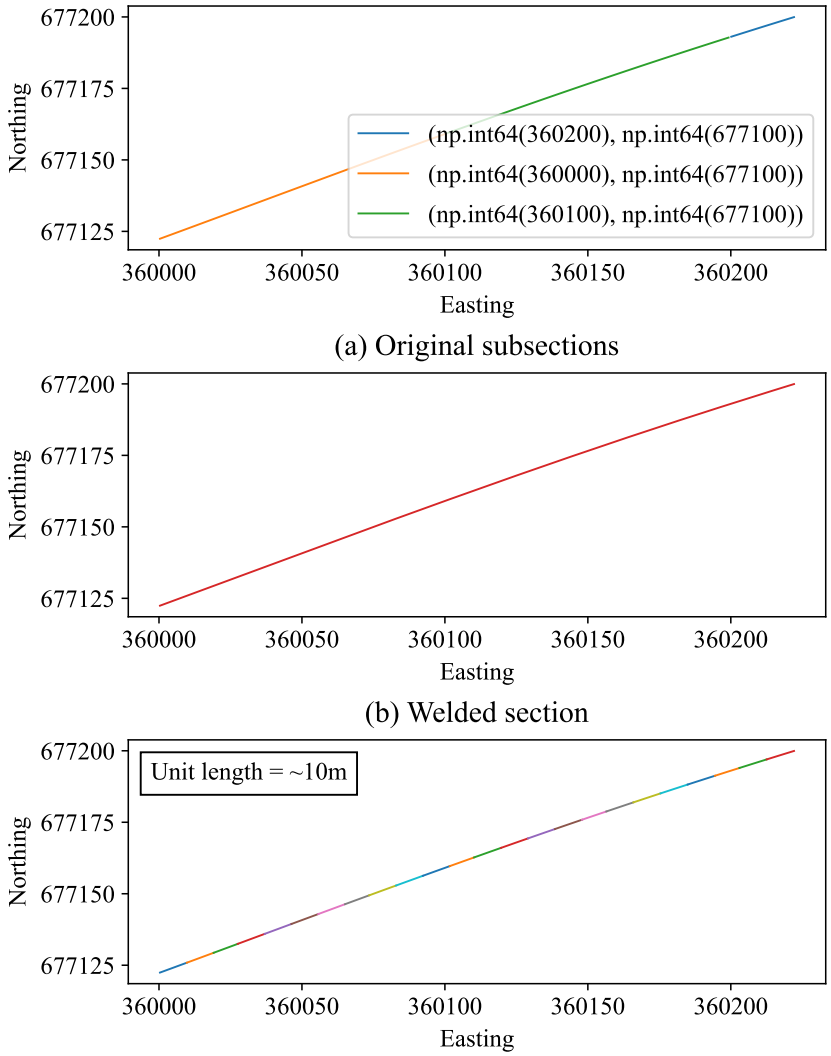
```
11
>>> for sub in ult_201910_subs.geoms:
...     print(sub.wkt[:50], "... ~%.4fm" % sub.length)
LINESTRING Z (340599.367 674259.012 32.211, 340598 ... ~9.9989m
LINESTRING Z (340589.994 674255.53 32.203, 340589. ... ~8.9999m
LINESTRING Z (340581.557 674252.397 32.2, 340580.6 ... ~9.9995m
LINESTRING Z (340572.183 674248.916 32.203, 340571 ... ~9.9990m
LINESTRING Z (340562.811 674245.431 32.208, 340561 ... ~9.0000m
LINESTRING Z (340554.375 674242.295 32.221, 340553 ... ~9.9992m
LINESTRING Z (340545.002 674238.812 32.246, 340544 ... ~9.0000m
LINESTRING Z (340536.566 674235.676 32.271, 340535 ... ~10.0005m
LINESTRING Z (340527.192 674232.192 32.296, 340526 ... ~9.9990m
LINESTRING Z (340517.82 674228.7070000001 32.323,  ... ~9.0000m
LINESTRING Z (340509.384 674225.571 32.348, 340508 ... ~9.9999m
```

Example 2 - Multiple tiles:

```
>>> tile_xy_list = [(360000, 677100), (360100, 677100), (360200, 677100)]
>>> # Right top of rail in the down direction within the above tiles in April␣
 ↪2020
>>> krdz_drt_multitiles_202004 = tm.load_classified_krdz(
...     tile_xy=tile_xy_list, pcd_date='202004', direction='down', element=
 ↪'right top')
>>> krdz_drt_multitiles_202004
   Year  Month  Tile_X  Tile_Y Direction         Element                    ␣
 ↪geometry
0  2020      4  360200  677100      Down  RightTopOfRail  LINESTRING Z (360200.
 ↪16 6...
1  2020      4  360000  677100      Down  RightTopOfRail  LINESTRING Z (360000.
 ↪349 ...
2  2020      4  360100  677100      Down  RightTopOfRail  LINESTRING Z (360100.
 ↪773 ...
>>> # Sum of the lengths of the three subsections
>>> krdz_drt_multitiles_202004['geometry'].map(lambda ls: ls.length).sum()
232.95245581071197
>>> # Weld the three subsections into a single section
>>> drt_202004 = weld_subsections(krdz_drt_multitiles_202004, start=(360000,␣
 ↪677100))
>>> # Total length (where the additional ~2 metres are from the two "joints")
>>> drt_202004.length
234.9511338346051
>>> # Divide the welded section into subsections of ~10 metres each
>>> drt_202004_subs = tm.split_section(section=drt_202004, unit_length=10)
>>> len(drt_202004_subs.geoms)
24
>>> for sub in drt_202004_subs.geoms:
...     print(sub.wkt[:50], "... ~%.4fm" % sub.length)
LINESTRING Z (360000.349 677122.4300000001 28.067, ... ~10.0003m
LINESTRING Z (360009.735 677125.8810000001 27.993, ... ~10.0008m
LINESTRING Z (360019.116 677129.347 27.935, 360020 ... ~8.9996m
LINESTRING Z (360027.556 677132.471 27.88, 360028. ... ~9.9980m
    ...                                           ...
LINESTRING Z (360175.425 677185.098 27.27, 360176. ... ~9.9974m
LINESTRING Z (360184.929 677188.2 27.258, 360185.8 ... ~9.9965m
LINESTRING Z (360194.445 677191.262 27.251, 360195 ... ~8.9960m
LINESTRING Z (360203.018 677193.988 27.248, 360203 ... ~9.9978m
LINESTRING Z (360212.557 677196.982 27.244, 360213 ... ~9.9976m
```

Illustration:

```
import matplotlib.pyplot as plt
import matplotlib.gridspec as gs
from matplotlib.offsetbox import AnchoredText
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg', font_name='Times New Roman')

fig = plt.figure(figsize=(7, 8), constrained_layout=True)
mgs = gs.GridSpec(3, 1, figure=fig)

colours = plt.get_cmap('tab10').colors

# Original separate subsections
ax1 = fig.add_subplot(mgs[0, :], aspect='equal', adjustable='box')
for i in krdz_drt_multitiles_202004.index:
    ls = krdz_drt_multitiles_202004.geometry[i]
    ls_xs, ls_ys = ls.coords.xy
    tile_xy = tuple(krdz_drt_multitiles_202004.loc[i, ['Tile_X', 'Tile_Y']].
 ↪values)
    ax1.plot(ls_xs, ls_ys, linewidth=1, color=colours[i], label=f'{tile_xy}')
ax1.set_xlabel('Easting', fontsize=13, labelpad=5)
ax1.set_ylabel('Northing', fontsize=13, labelpad=5)
ax1.set_title('(a) Original subsections', y=0, pad=-55)
ax1.legend()

# Welded section
ax2 = fig.add_subplot(mgs[1, :], aspect='equal', adjustable='box')
welded_xs, welded_ys = drt_202004.coords.xy
ax2.plot(welded_xs, welded_ys, linewidth=1, color=colours[i + 1])
ax2.set_xlabel('Easting', fontsize=13, labelpad=5)
ax2.set_ylabel('Northing', fontsize=13, labelpad=5)
ax2.set_title('(b) Welded section', y=0, pad=-55)

# Unit subsections
ax3 = fig.add_subplot(mgs[2, :], aspect='equal', adjustable='box')
for unit_sub in drt_202004_subs.geoms:
    sub_xs, sub_ys = unit_sub.coords.xy
    ax3.plot(sub_xs, sub_ys, linewidth=1)
anchored_text = AnchoredText('Unit length = ~10m', loc=2)
ax3.add_artist(anchored_text)
ax3.set_xlabel('Easting', fontsize=13, labelpad=5)
ax3.set_ylabel('Northing', fontsize=13, labelpad=5)
ax3.set_title('(c) Unit subsections', y=0, pad=-55)

# from pyhelpers.store import save_figure
# fig_pathname = "docs/source/_images/tm_split_section_demo"
# save_figure(fig, f"{fig_pathname}.svg", verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", verbose=True)
```



(a) Original subsections

(b) Welded section

(c) Unit subsections

Figure 89: Welded section and unit subsections of the top of the right rail of a track section in the down direction within the tiles (360000, 677100), (360100, 677100) and (360200, 677100).

TrackMovement.view_heatmap

`TrackMovement.`**`view_heatmap`**(*element*, *direction*, *subsect_len=100*, *pcd_dates=None*, *col_names=None*, *scale=1000*, *open_html=True*, *update=False*, ***kwargs*)

Create a heat map view of the track movement.

Parameters

- **`element`** (*`str | list | None`*) – Element of rail head, e.g. left/right top of rail or running edge; defaults to `None`.
- **`direction`** (*`str | list | None`*) – Railway direction, e.g. up and down directions; defaults to `None`.

- **subsect_len** (*int*) – Length (in metre) of a subsection for which movement is calculated; defaults to `10`.
- **pcd_dates** (*list | tuple | None*) – Dates of the point cloud data to compare; defaults to `None`.
- **col_names** (*list | None*) – Names of columns to be viewed.
- **scale** (*int | float*) – Scale of the calculated values of displacements on the heat map; defaults to `10 ** 3`.
- **open_html** (*bool*) – Whether to open a local HTML file in an explorer to view the map; defaults to `True`.
- **update** (*bool*) – Whether to reprocess the original data file(s); defaults to `False`.
- **kwargs** – [Optional] parameters of the method `load_movement()`.

Examples:

```
>>> from src.shaft import TrackMovement
>>> tm = TrackMovement()
>>> # tm.view_heatmap(element='Left Top', direction='Up', subsect_len=10,␣
 ↪update=True)
>>> tm.view_heatmap(element='Left Top', direction='Up', subsect_len=10)
```

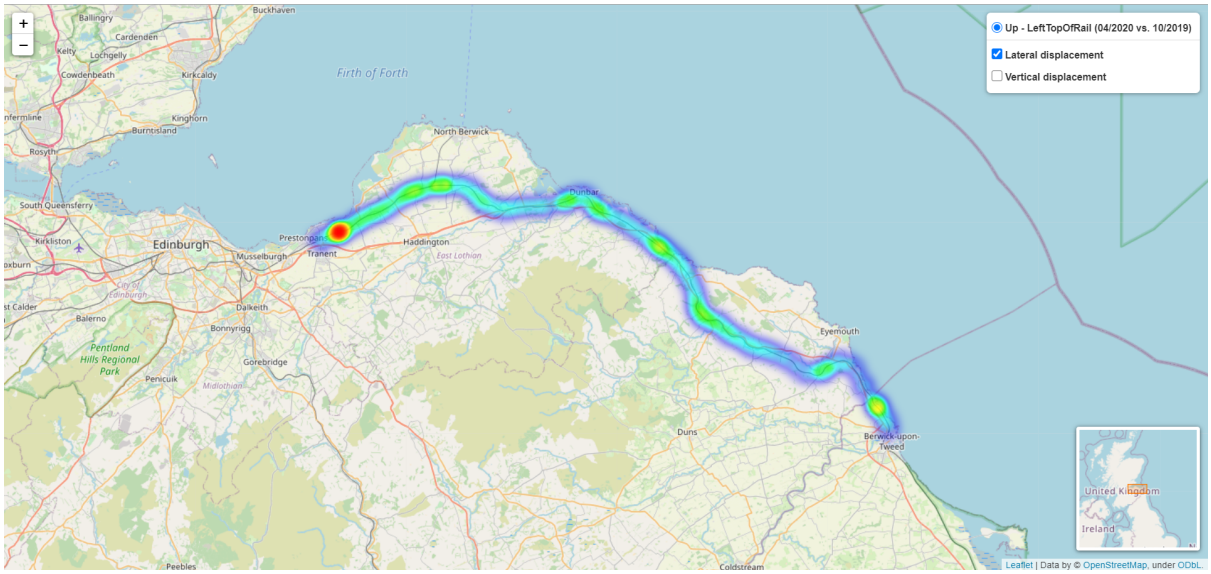


Figure 90: Heatmap for movement of the top of the left rail in the up direction (on a 10-metre basis).

TrackMovement.view_movement_violin_plot

static TrackMovement.**view_movement_violin_plot**(*data*, *fig_size=(10, 6)*, *save_as=None*, *dpi=600*, *verbose=False*, ***kwargs*)

Create a violin plot of the track movement.

Parameters

- **data** (*pandas.DataFrame*) – Data of the track movement.
- **fig_size** (*tuple[float, float]*) – Figure size; defaults to `(10, 6)`.
- **save_as** (*str | list | None*) – File format that the figure is saved as; defaults to `None`.
- **dpi** (*int | None*) – DPI for saving image; defaults to `600`.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to `False`.
- **kwargs** – [Optional] additional parameters of the function pyhelpers.store.save_figure.

Examples:

```
>>> from src.shaft import TrackMovement
>>> from pyhelpers.settings import mpl_preferences
>>> mpl_preferences(backend='TkAgg')
>>> tm = TrackMovement()
>>> movement_data_dict = tm.load_movement(
...     element='Left Top', direction='Up', subsect_len=10)
>>> movement_data = movement_data_dict['Up_LeftTopOfRail_201910_202004']
>>> column_names = ['lateral_displacement_mean', 'vertical_displacement_mean']
>>> data = movement_data[column_names] * 1000
>>> tm.view_movement_violin_plot(data, fig_size=(10, 6))
>>> # tm.view_movement_violin_plot(data, save_as=".svg", verbose=True)
```

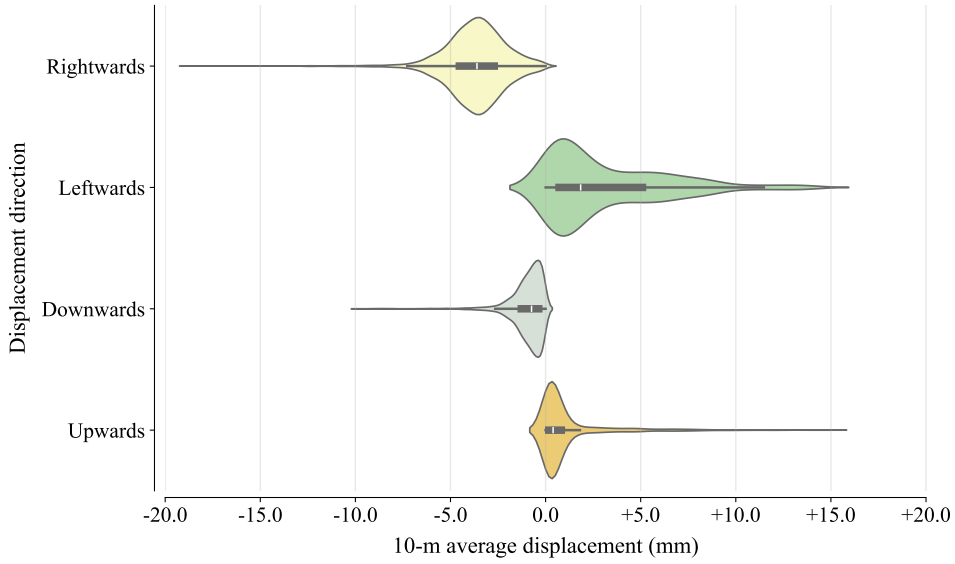


Figure 91: Violin plot of the average track movement for every 10-m section.

TrackMovement.weld_classified_krdz

TrackMovement.**weld_classified_krdz**(*element*, *direction*, *pcd_date*, *tile_xy=None*)

Weld the classified KRDZ rail head data for all tiles.

Parameters

- **element** (*str*) – Element of rail head.
- **direction** (*str*) – Railway direction; options include {'up', 'down'}.
- **pcd_date** (*str | int*) – Date of the point cloud data.
- **tile_xy** (*tuple | list | str | None*) – Easting and northing of a tile for the point cloud data; defaults to None.

Returns

Polyline geometry of the target track.

Return type

LineString

Examples:

```
>>> from src.shaft import TrackMovement
>>> tm = TrackMovement()
>>> element = 'Left Top'
>>> direction = 'Up'
>>> welded_ult_201910 = tm.weld_classified_krdz(element, direction, pcd_date=
↪'201910')
>>> welded_ult_201910.length
76299.56835581265
>>> welded_ult_202004 = tm.weld_classified_krdz(element, direction, pcd_date=
```

(continues on next page)

(continued from previous page)

```
↪'202004')
>>> welded_ult_202004.length
76300.69543101029
```

Illustration:

```
import matplotlib.pyplot as plt
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg', font_name='Times New Roman')

fig = plt.figure(figsize=(11, 5), constrained_layout=True)
ax = fig.add_subplot(aspect='equal', adjustable='box')
ax.grid()

xs_o, ys_o = welded_ult_201910.coords.xy
ax.plot(xs_o, ys_o, lw=1, marker='o', markersize=5, label='LeftTopOfRail 201910
↪')
xs_s, ys_s = welded_ult_202004.coords.xy
ax.plot(xs_s, ys_s, lw=1, marker='^', markersize=5, label='LeftTopOfRail 202004
↪')

ax.legend()

# from pyhelpers.store import save_figure
# fig_pathname_ = "docs/source/_images/tm_weld_classified_krdz_ult_demo"
# save_figure(fig, f"{fig_pathname_}.svg", transparent=True, verbose=True)
# save_figure(fig, f"{fig_pathname_}.pdf", transparent=True, verbose=True)
```

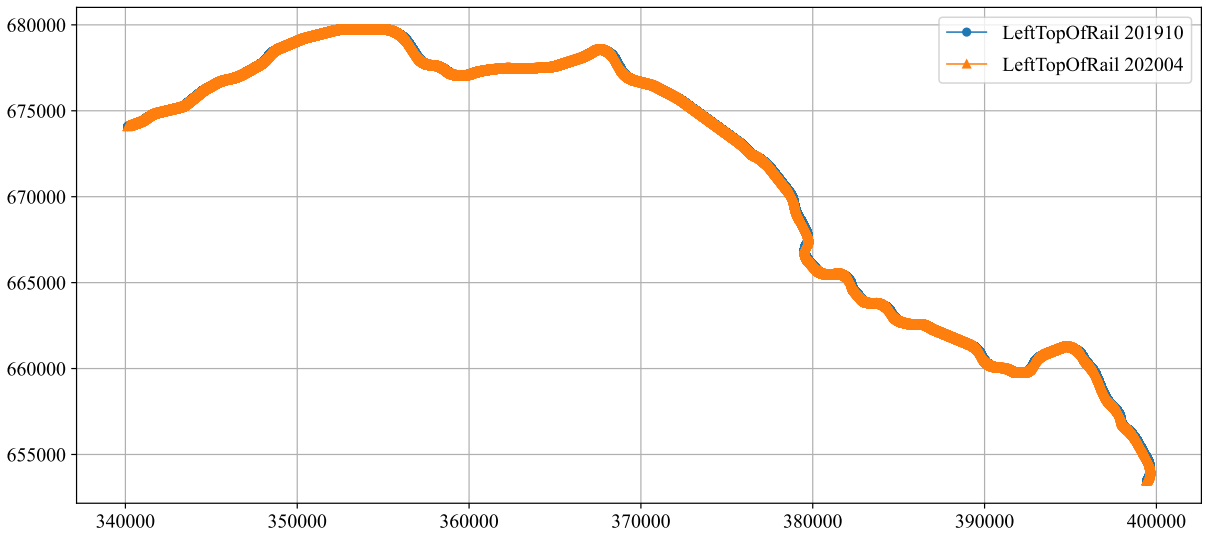


Figure 92: The top of left rail in the up direction (10/2019 vs. 04/2020).

FeatureCollator

class src.shaft.**FeatureCollator**(*elr=None*, *db_instance=None*, *verbose=True*, ***kwargs*)

Collate data on various features for developing a machine learning model to predict track movement (i.e. track fixity parameters).

Parameters

- **db_instance** (*TrackFixityDB | None*) – PostgreSQL database instance; defaults to None
- **elr** (*str | list | None*) – Engineer's Line Reference(s); defaults to None.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to True.

Variables

- **db_instance** (*pyhelpers.dbms.PostgreSQL*) – PostgreSQL database instance.
- **elr** (*str | list*) – Engineer's Line Reference(s).
- **ballast** (Ballast) – An instance of *Ballast*.
- **carrs** (CARRS) – An instance of *CARRS*.
- **cnm** (CNM) – An instance of *CNM*.
- **geology** (Geology) – An instance of *Geology*.
- **inm** (INM) – An instance of *INM*.
- **opas** (OPAS) – An instance of *OPAS*.
- **pcd** (PCD) – An instance of *PCD*.
- **track** (Track) – An instance of *Track*.
- **pseudo_mileage_dict** (*dict*) – Pseudo mileages for all available track IDs; defaults to None.
- **waymarks** (*pandas.DataFrame*) – Data of waymarks (with pseudo mileages); defaults to None.
- **ballast_summary** (*pandas.DataFrame*) – Data of ballast summary (with pseudo mileages); defaults to None.
- **ballast_features** (*pandas.DataFrame*) – Features collated from ballast summary for the pseudo mileages; defaults to None.
- **structures** (*pandas.DataFrame*) – Data of structure presence (with pseudo mileages); defaults to None.
- **inm_combined_data_report** (*pandas.DataFrame*) – INM combined data report (with pseudo mileages); defaults to None.

- **geological_features** (*pandas.DataFrame*) – Data of geology (with pseudo mileages); defaults to None.
- **track_quality** (*pandas.DataFrame*) – Data of track quality (with pseudo mileages); defaults to None.
- **subsection_buffer** (*pandas.Series*) – Buffers created for each track subsection; defaults to None.
- **subsection_nodes** (*pandas.DataFrame*) – Nodes (incl. centroid and two ends) for each track subsection; defaults to None.
- **element** (*str | list | None*) – Element of rail head, e.g. left/right top of rail or running edge; defaults to None.
- **direction** (*str | list | None*) – Railway direction, e.g. up and down directions; defaults to None.
- **subsect_len** (*int | None*) – Length (in metre) of a subsection for which movement is calculated; defaults to None.
- **track_movement** (*pandas.DataFrame*) – Data of the track movement with collated features.

Examples:

```
>>> from src.shaft import FeatureCollator
>>> fc = FeatureCollator(elr='ECM8')
Initialising the feature collator ... Done.
>>> fc.waymarks_
      ELR  ...                                    pseudo_geometry
0    ECM8  ...  LINESTRING Z (326301.33949999977 673958.802100...
1    ECM8  ...  LINESTRING Z (327289.98450000025 674251.033500...
2    ECM8  ...  LINESTRING Z (327696.2715999996 674194.5047999...
3    ECM8  ...  LINESTRING Z (328097.8353000004 674240.6223000...
4    ECM8  ...  LINESTRING Z (328494.3197999997 674324.2128999...
..    ...  ...                                                ...
210  ECM8  ...  LINESTRING Z (396983.12600000016 658556.672000...
211  ECM8  ...  LINESTRING Z (397172.99619999994 658203.952700...
212  ECM8  ...  LINESTRING Z (397435.50569999963 657906.502000...
213  ECM8  ...  LINESTRING Z (397718.7960000001 657613.5120999...
214  ECM8  ...  LINESTRING Z (397819.31799999997 657439.910499...
[215 rows x 6 columns]
```

Illustration:

```
import matplotlib.pyplot as plt
import numpy as np
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg', font_name='Times New Roman')

fig = plt.figure(figsize=(11, 5))
ax = fig.add_subplot(aspect='equal', adjustable='box')

for g in fc.waymarks_.pseudo_geometry:
    g = np.array(g.coords)
```

(continues on next page)

(continued from previous page)

```
    ax.plot(g[:, 0], g[:, 1])

ax.set_xlabel('Easting', fontsize=14)
ax.set_ylabel('Northing', fontsize=14)

fig.tight_layout()

# from pyhelpers.store import save_figure
# fig_filename = "fc_pseudo_waymarks_demo"
# save_figure(fig, f"docs\source\_images\{fig_filename}.svg", verbose=True)
# save_figure(fig, f"docs\source\_images\{fig_filename}.pdf", verbose=True)
```

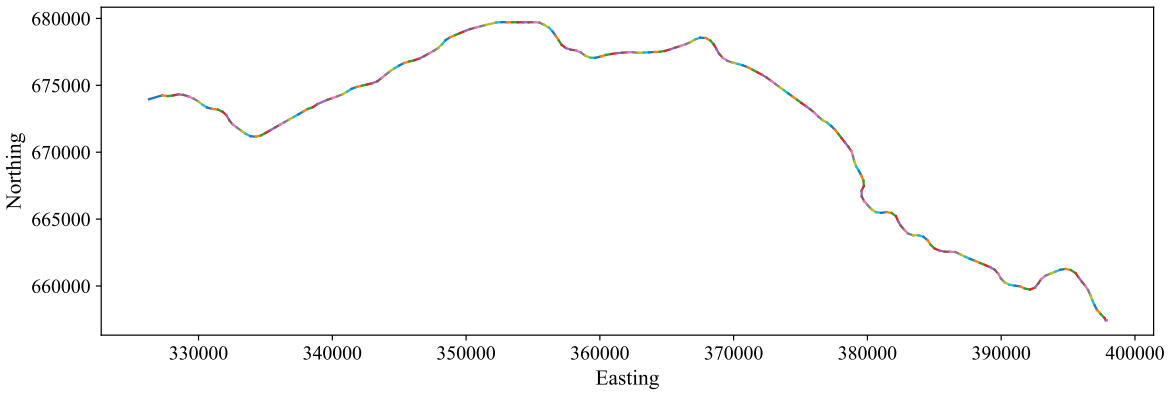


Figure 93: Pseudo waymarks.

Note: No directly defined attributes. See inherited class attributes.

Methods:

assign_pseudo_mileage(elr, data[, tid_col, ...])	Assign pseudo mileage to cases in a given data set.
assign_pseudo_mileage_to_ballast([elr, ...])	Assign pseudo mileage to the data of ballast summary.
assign_pseudo_mileage_to_inm_cdr([elr, ...])	Assign pseudo mileage to the INM combined data report.
assign_pseudo_mileage_to_track_quality([...])	Assign pseudo mileage to the preprocessed track quality data.
calculate_track_quality_stats([elr, ...])	Calculate basic statistics of track quality data.
collate_ballast_features([elr, ret_dat])	Collate feature data from the data of ballast summary (for modelling).
collate_nearest_structure(structure_data, ...)	Collate information about the presence of a certain structure at each subsection (for which track movement is calculated).
find_nearest_waymark_for_subsections([...])	Find the nearest waymark for each track subsection.
get_subsection_centroid_and_ends(subsect_geom)	Get the centroid and two ends of every subsection.
integrate_data(element, direction, subsect_len)	Construct an integrated data set for the development of a machine learning model for track fixity.
make_mileage_sequence(dat, start_mil_col[, ...])	Put together all values of the mileages into a sequence.
make_subsection_buffer(subsect_geom[, ...])	Make a buffer for every subsection for which track movement is calculated.
view_subsection_buffer([buf_type, buf_dist, ...])	View subsection buffer.

FeatureCollator.assign_pseudo_mileage

FeatureCollator.**assign_pseudo_mileage**(*elr*, *data*, *tid_col='TID'*, *start_mil_col='StartMileage'*, *end_mil_col='EndMileage'*, *pseudo_mil_dict=None*, *suppl_mil_data=None*, *suppl_mil_col='Mileage'*)

Assign pseudo mileage to cases in a given data set.

Parameters

- **elr** (*str*) – Engineer's Line Reference.
- **data** (*pandas.DataFrame*) – A data set of one resource of the

project.

- **tid_col** (*str*) – Name of the column for track IDs; defaults to `'TID'`.
- **start_mil_col** (*str*) – Name of the column for start mileage; defaults to `'StartMileage'`.
- **end_mil_col** (*str*) – Name of the column for end mileage; defaults to `'EndMileage'`.
- **suppl_mil_data** (*dict | pandas.DataFrame | None*) – A supplementary data set about mileages; defaults to `None`.
- **suppl_mil_col** (*str*) – Column name for the mileage data in `suppl_mil_data`; defaults to `'Mileage'`.
- **pseudo_mil_dict** (*dict | None*) – Data of pseudo mileages; when `pseudo_mil_dict=None` (default), it retrieves the data by using the method *Track.load_pseudo_mileage_dict()*.

Returns

Processed data set with assigned pseudo mileages.

Return type

pandas.DataFrame

Examples:

```
>>> from src.shaft import FeatureCollator
>>> test_elr = 'ECM8'
>>> fc = FeatureCollator(elr=test_elr)
Initialising the feature collator ... Done.
>>> ballast_summary = fc.ballast.load_data(elr=test_elr)
>>> ballast_summary.head()
       ID  GEOGIS Switch ID Track priority  ...   BCF StartMileage EndMileage
0  225960              NaN  Running lines  ...  1.50       0.1252     0.1291
1  225961          23481.0  Running lines  ...  1.00       0.1291     0.1320
2  225962          23481.0  Running lines  ...  1.00       0.1320     0.1350
3  225963              NaN  Running lines  ...  1.47       0.1350     0.1367
4  225964              NaN  Running lines  ...  1.03       0.1367     0.1368
[5 rows x 60 columns]
>>> ballast_summary_ = fc.assign_pseudo_mileage(elr=test_elr, data=ballast_
 ↪summary)
>>> ballast_summary_.head()
       ID  ...                                     pseudo_geometry
0  225960  ...  LINESTRING Z (326843.3266799995 674149.1670249...
1  225961  ...  LINESTRING Z (326876.7641107336 674161.6463143...
2  225962  ...  LINESTRING Z (326901.8077764853 674171.0149599...
3  225963  ...  LINESTRING Z (326927.8255407451 674180.4099819...
4  225964  ...  LINESTRING Z (326942.5269018637 674185.8494323...
[5 rows x 63 columns]
```

Illustration:

```
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg', font_name='Times New Roman')

fig = plt.figure(figsize=(11, 5))
ax = fig.add_subplot(aspect='equal', adjustable='box')

for g in ballast_summary_.pseudo_geometry:
    g_ = np.array(g.coords)
    ax.plot(g_[:, 0], g_[:, 1])

ax.set_xlabel('Easting', fontsize=14, labelpad=10)
ax.set_ylabel('Northing', fontsize=14, labelpad=10)

fig.tight_layout()

# from pyhelpers.store import save_figure
# fig_filename = "fc_assign_pseudo_mileage_demo"
# save_figure(fig, f"docs\source\_images\{fig_filename}.svg", verbose=True)
# save_figure(fig, f"docs\source\_images\{fig_filename}.pdf", verbose=True)
```

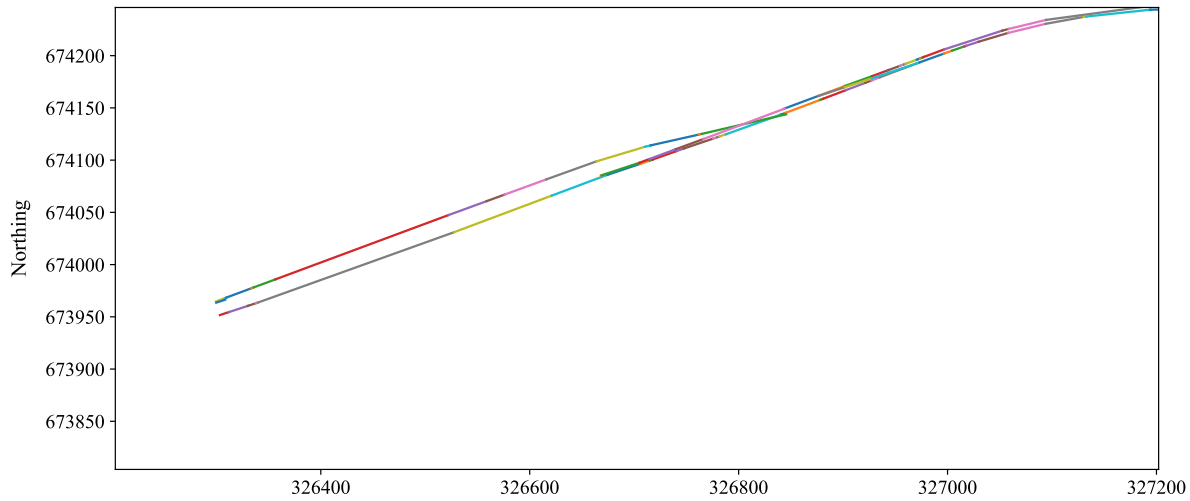


Figure 94: Pseudo mileages for a subset of the data of ECM8 ballast summary.

FeatureCollator.assign_pseudo_mileage_to_ballast

FeatureCollator.**assign_pseudo_mileage_to_ballast**(*elr=None*, *pseudo_mil_dict=None*, *ret_dat=False*)

Assign pseudo mileage to the data of ballast summary.

Parameters

- **elr** (*str | list | None*) – Engineer's Line Reference(s); defaults to None.

- **pseudo_mil_dict** (*dict | None*) – Data of pseudo mileages; when pseudo_mil_dict=None (default), it retrieves the data by using the method *Track.load_pseudo_mileage_dict()*.
- **ret_dat** (*bool*) – Whether to return the processed data; defaults to False.

Returns
Data of ballast summary with assigned pseudo mileages.

Return type
pandas.DataFrame

Examples:

```
>>> from src.shaft import FeatureCollator
>>> fc = FeatureCollator(elr=['ECM7', 'ECM8'])
Initialising the feature collator ... Done.
>>> fc.assign_pseudo_mileage_to_ballast()
>>> fc.ballast_summary
          ID  GEOGIS Switch ID Track priority  ...   BCF StartMileage EndMileage
0     221491               NaN  Running lines  ...  1.27       0.0000     0.0054
1     221492               NaN  Running lines  ...  1.27       0.0054     0.0060
2     221493               NaN  Running lines  ...  1.27       0.0060     0.0066
3     221494               NaN  Running lines  ...  1.28       0.0066     0.0084
4     221495               NaN  Running lines  ...  1.28       0.0084     0.0092
      ...               ...           ...  ...   ...          ...        ...
8179  230311               NaN  Running lines  ...  1.29       3.0694     3.0748
8180  230312               NaN  Running lines  ...  1.29       3.0748     3.0749
8181  230313               NaN  Running lines  ...  1.29       3.0749     3.0777
8182  230314               NaN  Running lines  ...  1.29       3.0777     3.0832
8183  230315           24283.0  Running lines  ...  1.20       3.0832     3.0869
[8184 rows x 60 columns]
>>> fc.ballast_summary_
          ID  ...                                    pseudo_geometry
0     221491  ...  LINESTRING Z (424617.6578774111 563817.2014027...
1     221492  ...  LINESTRING Z (424666.8570424306 563821.3953917...
2     221493  ...  LINESTRING Z (424672.446802165 563821.87189151...
3     221494  ...  LINESTRING Z (424678.0365618993 563822.3483912...
4     221495  ...  LINESTRING Z (424694.8074982702 563823.7573691...
      ...  ...                                                 ...
8179  230311  ...  LINESTRING Z (330859.5571851924 673253.5898562...
8180  230312  ...  LINESTRING Z (330907.5838895333 673241.5453524...
8181  230313  ...  LINESTRING Z (330908.473272947 673241.32230603...
8182  230314  ...  LINESTRING Z (330933.3171053747 673234.8605545...
8183  230315  ...  LINESTRING Z (330981.5182506089 673220.0618065...
[8180 rows x 63 columns]
>>> fc.ballast_summary_[fc.ballast_summary_.columns[-3:]]
                  pseudo_geometry_start  ...                          pseudo_
↪geometry
0     POINT Z (424617.6578774138 563817  ...  LINESTRING Z (424617.6578774138␣
↪56381...
1     POINT Z (424666.8570424306 563821  ...  LINESTRING Z (424666.8570424306␣
↪56382...
2      POINT Z (424672.446802165 563821  ...  LINESTRING Z (424672.446802165␣
↪563821...
3     POINT Z (424678.0365618993 563822  ...  LINESTRING Z (424678.0365618993␣
↪56382...
```

(continues on next page)

(continued from previous page)

```
4     POINT Z (424694.8074982702 563823  ...  LINESTRING Z (424694.8074982702␣
↪56382...
                                        ...  ...                              ␣
↪   ...
8179  POINT Z (330859.5571851924 673253  ...  LINESTRING Z (330859.5571851924␣
↪67325...
8180  POINT Z (330907.5838895333 673241  ...  LINESTRING Z (330907.5838895333␣
↪67324...
8181    POINT Z (330908.473272947 67324  ...  LINESTRING Z (330908.473272947␣
↪673241...
8182  POINT Z (330933.3171053747 673234  ...  LINESTRING Z (330933.3171053747␣
↪67323...
8183  POINT Z (330981.5182506089 673220  ...  LINESTRING Z (330981.5182506089␣
↪67322...
[8180 rows x 3 columns]
```

See also:

- Examples for the method `assign_pseudo_mileage()`.

FeatureCollator.assign_pseudo_mileage_to_inm_cdr

`FeatureCollator.`**`assign_pseudo_mileage_to_inm_cdr`**(*elr=None*, *pseudo_mil_dict=None*, *ret_dat=False*)

Assign pseudo mileage to the INM combined data report.

Parameters

- **elr** (*str | list | None*) – Engineer's Line Reference(s); defaults to `None`.
- **pseudo_mil_dict** (*dict | None*) – Data of pseudo mileages; when `pseudo_mil_dict=None` (default), it retrieves the data by using the method `Track.load_pseudo_mileage_dict()`.
- **ret_dat** (*bool*) – Whether to return the processed data; defaults to `False`.

Returns

INM combined data report with pseudo mileages.

Return type

pandas.DataFrame

Examples:

```
>>> from src.shaft import FeatureCollator
>>> fc = FeatureCollator(elr=['ECM7', 'ECM8'])
Initialising the feature collator ... Done.
>>> fc.assign_pseudo_mileage_to_inm_cdr()
>>> fc.inm_combined_data_report.head()
           RTE_NAME RTE_ORG_CODE  ... RELAYINGPOLICY TRACK_POSITION
```

(continues on next page)

(continued from previous page)

```
0  London North East           QG  ...              0            5.0
1  London North East           QG  ...              0            7.0
2  London North East           QG  ...              0            6.0
3  London North East           QG  ...              0            4.0
4  London North East           QG  ...              0            3.0
[5 rows x 55 columns]
>>> fc.inm_combined_data_report_.head()
           RTE_NAME  ...                                  pseudo_geometry
0  London North East  ...  LINESTRING Z (424635.8603999997 563764.4418000...
1  London North East  ...  LINESTRING Z (424639.1655999999 563754.7467999...
2  London North East  ...  LINESTRING Z (424637.4572000001 563759.7326999...
3  London North East  ...  LINESTRING Z (424634.2836999996 563783.1506999...
4  London North East  ...  LINESTRING Z (424633.2932000002 563788.2982999...
[5 rows x 58 columns]
```

Illustration:

```
import numpy as np
import matplotlib.pyplot as plt
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg', font_name='Times New Roman')

fig = plt.figure(figsize=(7, 7), constrained_layout=True)
ax = fig.add_subplot(aspect='equal', adjustable='box')

for g in fc.inm_combined_data_report_['pseudo_geometry']:
    g_ = np.array(g.coords)
    ax.plot(g_[:, 0], g_[:, 1])

ax.set_xlabel('Easting', fontsize=14, labelpad=8.0)
ax.set_ylabel('Northing', fontsize=14, labelpad=8.0)

# from src.utils import cd_docs_source
# from pyhelpers.store import save_figure
# path_to_fig = cd_docs_source("_images\fc_assign_pseudo_mileage_to_inm_cdr_demo
 ↪")
# save_figure(fig, path_to_fig + ".svg", verbose=True)
# save_figure(fig, path_to_fig + ".pdf", verbose=True)
```

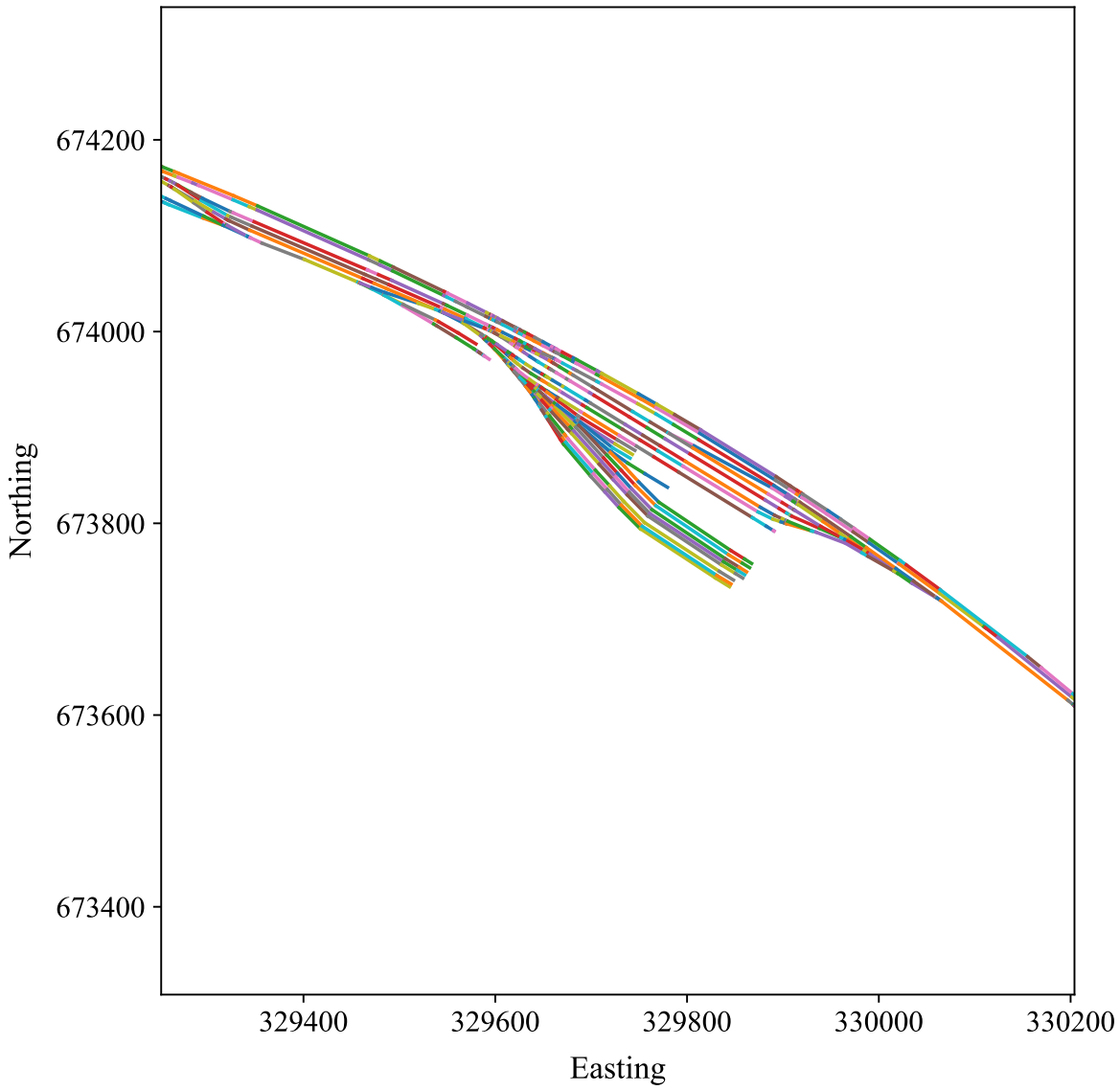


Figure 95: Pseudo mileages for ECM8 INM combined data report.

FeatureCollator.assign_pseudo_mileage_to_track_quality

FeatureCollator.**assign_pseudo_mileage_to_track_quality**(*elr=None*, *tq_date=None*, *agg_method=None*, *pseudo_mil_dict=None*, *ret_dat=False*)

Assign pseudo mileage to the preprocessed track quality data.

Parameters

- **elr** (*str | list | None*) – Engineer's Line Reference(s); defaults to None.
- **tq_date** (*list | None*) – Dates of track quality data; defaults to

None.

- **agg_method** – Method of aggregation; when agg_method=None (default), it calculates the average of grouped data; an alternative method is 'rms', i.e. 'root-mean-square'.
- **pseudo_mil_dict** (*dict | None*) – Data of pseudo mileages; when pseudo_mil_dict=None (default), it retrieves the data by using the method *Track.load_pseudo_mileage_dict()*.
- **ret_dat** (*bool*) – Whether to return the processed data; defaults to False.

Returns

Track quality data with assigned pseudo mileages.

Return type

pandas.DataFrame

Examples:

```
>>> from src.shaft import FeatureCollator
>>> fc = FeatureCollator(elr='ECM8')
Initialising the feature collator ... Done.
>>> fc.assign_pseudo_mileage_to_track_quality()
>>> fc.track_quality
          ELR  Track Id  ... Dipped Left (mrad) Dipped Right (mrad)
0        ECM8      2100  ...                0.0                 0.0
1        ECM8      2100  ...                0.0                 0.0
2        ECM8      1100  ...                0.0                 0.0
3        ECM8      1100  ...                0.0                 0.0
4        ECM8      2100  ...                0.0                 0.0
       ...       ...  ...                ...                 ...
1375425  ECM8      1100  ...                0.0                 0.0
1375426  ECM8      1100  ...                0.0                 0.0
1375427  ECM8      1100  ...                0.0                 0.0
1375428  ECM8      1100  ...                0.0                 0.0
1375429  ECM8      1100  ...                0.0                 0.0
[1375430 rows x 23 columns]
>>> fc.track_quality_
       ELR  ...                                    pseudo_geometry
0     ECM8  ...  POINT Z (340000.8967833333 674040.6199604174 0)
1     ECM8  ...    POINT Z (340019.6609500001 674047.63910625 0)
2     ECM8  ...  POINT Z (340038.4317759821 674054.6556251637 0)
3     ECM8  ...  POINT Z (340057.3424475812 674061.6169787806 0)
4     ECM8  ...  POINT Z (340076.2531191803 674068.5783323975 0)
    ...  ...                                                 ...
7021  ECM8  ...  POINT Z (397758.8943053244 657528.7732021383 0)
7022  ECM8  ...  POINT Z (397770.3546583527 657512.2525615764 0)
7023  ECM8  ...  POINT Z (397781.5221755836 657495.5327715443 0)
7024  ECM8  ...  POINT Z (397792.4366677299 657478.6479913326 0)
7025  ECM8  ...  POINT Z (397803.1121100634 657461.6104920406 0)
[7026 rows x 19 columns]
```

FeatureCollator.calculate_track_quality_stats

FeatureCollator.**calculate_track_quality_stats**(*elr=None*, *tq_date=None*, *agg_method=None*)

Calculate basic statistics of track quality data.

Parameters

- **elr** (*str | list | None*) – Engineer's Line Reference(s); defaults to None.
- **tq_date** (*list | None*) – Dates of track quality data; defaults to None.
- **agg_method** – Method of aggregation; when agg_method=None (default), it calculates the average of grouped data; an alternative method is 'rms', i.e. 'root-mean-square'.

Returns

Basic statistics of track quality data.

Return type

pandas.DataFrame

Examples:

```
>>> from src.shaft import FeatureCollator
>>> fc = FeatureCollator(elr='ECM8')
Initialising the feature collator ... Done.
>>> tq_stats_data = fc.calculate_track_quality_stats()
>>> tq_stats_data
       ELR  Track Id  ...  Dipped Left (mrad)  Dipped Right (mrad)
0     ECM8      1100  ...                 0.0               0.0000
1     ECM8      1100  ...                 0.0               0.0000
2     ECM8      1100  ...                 0.0               0.0000
3     ECM8      1100  ...                 0.0              -0.4313
4     ECM8      1100  ...                 0.0              -1.8274
    ...       ...  ...                 ...                  ...
7021  ECM8      2100  ...                 0.0               0.0000
7022  ECM8      2100  ...                 0.0               0.0000
7023  ECM8      2100  ...                 0.0               0.0000
7024  ECM8      2100  ...                 0.0               0.0000
7025  ECM8      2100  ...                 0.0               0.0000
[7026 rows x 18 columns]
```

FeatureCollator.collate_ballast_features

FeatureCollator.**collate_ballast_features**(*elr=None*, *ret_dat=False*, ***kwargs*)

Collate feature data from the data of ballast summary (for modelling).

Parameters

- **elr** (*str | list | None*) – Engineer's Line Reference(s); defaults to None.

- **ret_dat** (*bool*) – Whether to return the processed data; defaults to `False`.
- **kwargs** – [Optional] parameters of the method `TrackMovement.load_movement()`.

Returns
Data (basic statistics) of ballast summary for modelling.

Return type
pandas.DataFrame

Attention: Currently, the features include only curvature, cant, maximum speed and maximum axle load.

Examples:

```
>>> from src.shaft import FeatureCollator
>>> fc = FeatureCollator(elr=['ECM7', 'ECM8'])
>>> fc.collate_ballast_features(element='Left Top', direction='Up', subsect_
 ↪len=1000)
>>> fc.ballast_features
                                                  Curvature  ...  Max axle␣
 ↪load
subsection                                                   ...
LINESTRING Z (399428.96 653473.9 34.442, 399429...  -0.000849  ...            ␣
 ↪26
LINESTRING Z (399611.8618926046 654427.63984359...  -0.000849  ...            ␣
 ↪26
LINESTRING Z (399170.8610524134 655321.58516764...  -0.000337  ...            ␣
 ↪26
LINESTRING Z (398643.3364543988 656169.10746808...   0.001189  ...            ␣
 ↪26
LINESTRING Z (397987.7488072964 656909.70011834...  -0.001266  ...            ␣
 ↪26
                                                        ...  ...           ...
LINESTRING Z (344013.9924472624 675799.46466366...   0.001075  ...            ␣
 ↪26
LINESTRING Z (343206.4015371799 675215.92935916...   0.001075  ...            ␣
 ↪26
LINESTRING Z (342238.3969935408 674967.57560955...  -0.000416  ...            ␣
 ↪26
LINESTRING Z (341298.8661618733 674644.90478813...   0.000898  ...            ␣
 ↪26
LINESTRING Z (340414.7337679755 674190.36774200...   0.000000  ...            ␣
 ↪26
[77 rows x 4 columns]
>>> fc.ballast_features.index.name
'subsection'
>>> fc.ballast_features.columns.to_list()
['Curvature', 'Cant', 'Max speed', 'Max axle load']
```

FeatureCollator.collate_nearest_structure

FeatureCollator.**collate_nearest_structure**(*structure_data*, *structure_name*, *ret_dat=False*, ***kwargs*)

Collate information about the presence of a certain structure at each subsection (for which track movement is calculated).

Parameters

- **structure_data** (*pandas.DataFrame*) – Data of a structure.
- **structure_name** (*str*) – Name of a structure.
- **ret_dat** (*bool*) – Whether to return the processed data; defaults to False.
- **kwargs** – [Optional] parameters of the method TrackMovement.load_movement().

Returns

Count of the nearest asset to each subsection for which track movement is calculated.

Return type

dict

Examples:

```
>>> from src.shaft import FeatureCollator
>>> test_elr = 'ECM8'
>>> fc = FeatureCollator(elr=test_elr)
Initialising the feature collator ... Done.
>>> fc.track_movement = fc.load_movement(
...     element='Left Top', direction='Up', subsect_len=1000)
>>> type(fc.track_movement)
dict
>>> list(fc.track_movement.keys())
['Up_LeftTopOfRail_201910_202004']
>>> trk_movement_ = fc.track_movement['Up_LeftTopOfRail_201910_202004']
>>> trk_movement_.head()
                                    subsection  ...  vertical_displacement_b_
↪abs_max
0  LINESTRING Z (399428.96 653473.9 34.442, 3...  ...                      -0.
↪010023
1  LINESTRING Z (399611.8618926046 654427.639...  ...                      -0.
↪006940
2  LINESTRING Z (399170.8610524134 655321.585...  ...                      -0.
↪002362
3  LINESTRING Z (398643.3364543988 656169.107...  ...                      -0.
↪010806
4  LINESTRING Z (397987.7488072964 656909.700...  ...                      -0.
↪007337
[5 rows x 37 columns]
>>> ol_bdg = fc.carrs.load_overline_bridges_shp(elr=test_elr)
>>> ol_bdg.head()
  Entity  ...                                          geometry
```

(continues on next page)

(continued from previous page)

```
0  Point  ...  POINT Z (329001.3986999998 674251.2369999997 0)
1  Point  ...  POINT Z (329243.2949999999 674171.7059000004 0)
2  Point  ...  POINT Z (331516.5432000002 673158.5880999994 0)
3  Point  ...  POINT Z (331284.2423999999 673210.6504999995 0)
4  Point  ...  POINT Z (331820.4682999998 673006.2052999996 0)
[5 rows x 78 columns]
>>> fc.collate_nearest_structure(ol_bdg, structure_name='Overline bridges')
>>> fc.structures_['Overline bridges'].head()
                                                  Overline bridges
subsection
LINESTRING Z (399428.96 653473.9 34.442, 399429...                0
LINESTRING Z (399611.8618926046 654427.63984359...                0
LINESTRING Z (399170.8610524134 655321.58516764...                0
LINESTRING Z (398643.3364543988 656169.10746808...                0
LINESTRING Z (397987.7488072964 656909.70011834...                0
>>> fc.structures_['Overline bridges'].sum()
Overline bridges    40
dtype: int64
```

Illustration:

```
import numpy as np
import matplotlib.pyplot as plt
from shapely.geometry import MultiPolygon
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg', font_name='Times New Roman')

# Get track geometry
track_movement = list(fc.track_movement.values())[0]
sub_geom = track_movement.subsection
sub_buffers = fc.make_subsection_buffer(sub_geom, buf_type=1, buf_dist=None)

fig = plt.figure(figsize=(11, 5), constrained_layout=True)
ax = fig.add_subplot(aspect='equal', adjustable='box')

colours = plt.get_cmap('tab10').colors

for g in sub_geom:
    g = np.array(g.coords)
    ax.plot(g[:, 0], g[:, 1], color=colours[0])
ax.plot([], [], color=colours[0], label='Track section (length = 1 km)')

for buf in fc.subsection_buffer_cir:
    buf_ = np.array(buf.exterior.coords)
    ax.plot(buf_[:, 0], buf_[:, 1], color=colours[1])
ax.scatter(
    [], [], 100, marker='o', fc='none', ec=colours[1],
    label='Buffer zone (diameter = 1 km)')

# Select all overline bridges that are within the study area
convex_hull = MultiPolygon(fc.subsection_buffer_cir.to_list()).convex_hull
ol_bdg_geometry = [x for x in ol_bdg.geometry if x.within(convex_hull)]

for ob in ol_bdg_geometry:
    ax.scatter(ob.x, ob.y, color=colours[2], s=25)
```

(continues on next page)

(continued from previous page)

```
ax.scatter([], [], marker='o', color=colours[2], label='Overline bridge')

ax.tick_params(axis='both', which='major', labelsize=18)
ax.set_xlabel('Easting', fontsize=20, labelpad=8.0)
ax.set_ylabel('Northing', fontsize=20, labelpad=8.0)

ax.legend(loc='best', numpoints=1, ncol=1, fontsize=18)

# from pyhelpers.store import save_figure
# fig_pathname = "docs/source/_images/fc_collate_nearest_structure_demo"
# save_figure(fig, f"{fig_pathname}.svg", transparent=True, verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", transparent=True, verbose=True)
```

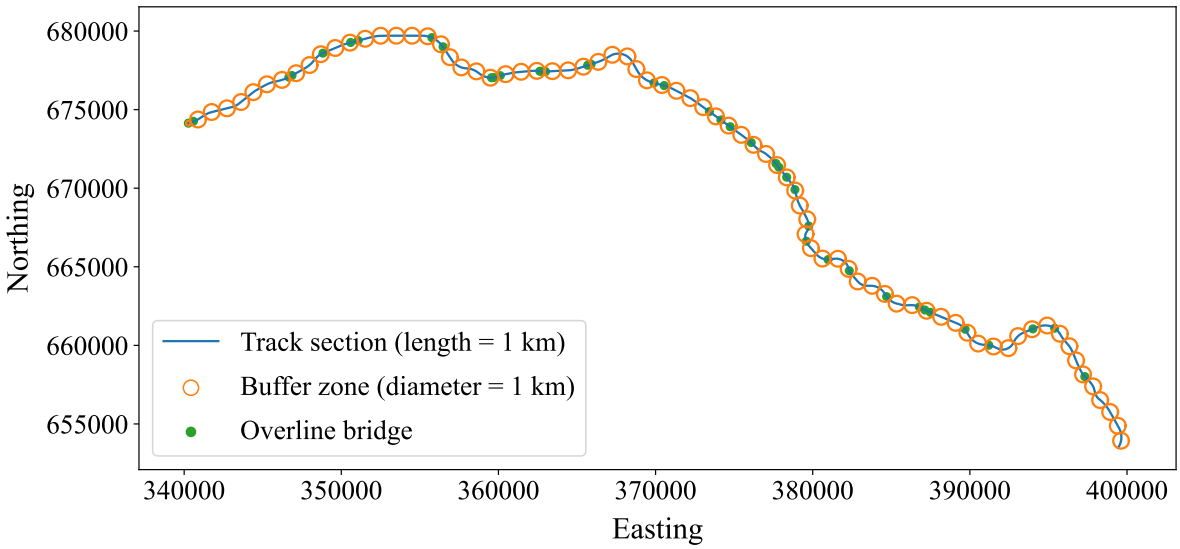


Figure 96: Overline bridges and buffers of the track subsections.

FeatureCollator.find_nearest_waymark_for_subsection

FeatureCollator.**find_nearest_waymark_for_subsection**(*subsect_geom*)

Find the nearest waymark for each track subsection.

Parameters

subsect_geom (*pandas.Series | list | numpy.ndarray | LineString*) – A sequence of geometry objects each representing a track subsection.

Returns

The nearest waymark for each track subsection.

Return type

pandas.DataFrame

Examples:

```
>>> from src.shaft import FeatureCollator
>>> fc = FeatureCollator(elr='ECM8')
Initialising the feature collator ... Done.
>>> trk_movement = fc.load_movement(
...     element='Left top', direction='Up', subsect_len=1000)
>>> trk_movement_ = list(trk_movement.values())[0]
>>> sub_geom = trk_movement_['subsection']
>>> sub_geom
0     LINESTRING Z (399428.96 653473.9 34.442, 39942...
1     LINESTRING Z (399611.8618926046 654427.6398435...
2     LINESTRING Z (399170.8610524134 655321.5851676...
3     LINESTRING Z (398643.3364543988 656169.1074680...
4     LINESTRING Z (397987.7488072964 656909.7001183...
                          ...
72    LINESTRING Z (344013.9924472624 675799.4646636...
73    LINESTRING Z (343206.4015371799 675215.9293591...
74    LINESTRING Z (342238.3969935408 674967.5756095...
75    LINESTRING Z (341298.8661618733 674644.9047881...
76    LINESTRING Z (340414.7337679755 674190.3677420...
Name: subsection, Length: 77, dtype: object
>>> nearest_waymark_for_subsect = fc.find_nearest_waymark_for_subsection(sub_
 ↪geom)
>>> nearest_waymark_for_subsect
    Waymark_StartMileage  ...                                Waymark_PseudoGeom
0                54.1107  ...  LINESTRING Z (397819.318 657439.9104999993 0, ...
1                54.1107  ...  LINESTRING Z (397819.318 657439.9104999993 0, ...
2                54.1107  ...  LINESTRING Z (397819.318 657439.9104999993 0, ...
3                54.1107  ...  LINESTRING Z (397819.318 657439.9104999993 0, ...
4                54.0440  ...  LINESTRING Z (397435.5056999996 657906.5020000...
..                   ...  ...                                                ...
72               12.0440  ...  LINESTRING Z (343370.9899000004 675301.2421000...
73               11.0880  ...  LINESTRING Z (342214.7589999996 674971.2629000...
74               11.0000  ...  LINESTRING Z (341443.9992000004 674738.0125999...
75               10.0440  ...  LINESTRING Z (340382.7313999999 674179.2847000...
76               10.0000  ...  LINESTRING Z (339999.9721999997 674043.0051000...
[77 rows x 3 columns]
```

FeatureCollator.get_subsection_centroid_and_ends

FeatureCollator.**get_subsection_centroid_and_ends**(*subsect_geom*)

Get the centroid and two ends of every subsection.

Parameters

subsect_geom (*pandas.Series | list | numpy.ndarray | LineString*) – A sequence of geometry objects each representing a track subsection.

Returns

The centroid and two ends of every track subsection.

Return type

pandas.DataFrame

Examples:

```
>>> from src.shaft import FeatureCollator
>>> fc = FeatureCollator(elr='ECM8')
Initialising the feature collator ... Done.
>>> trk_movement = fc.load_movement(
...     element='Left top', direction='Up', subsect_len=1000)
>>> trk_movement_ = list(trk_movement.values())[0]
>>> sub_geom = trk_movement_.subsection
>>> nodes = fc.get_subsection_centroid_and_ends(subsect_geom=sub_geom)
>>> nodes
                                subsect_a  ...                        subsect_
↪centroid
0                 POINT Z (399428.96 6534  ...  POINT Z (399621.74128885 653931.
↪27....
1   POINT Z (399611.8618926046 654427....  ...  POINT Z (399412.8613179412␣
↪654884.0...
2   POINT Z (399170.8610524134 655321....  ...  POINT Z (398927.8690315923␣
↪655758....
3   POINT Z (398643.3364543988 656169....  ...  POINT Z (398288.7644842083␣
↪656519....
4   POINT Z (397987.7488072964 656909....  ...  POINT Z (397844.7254107677␣
↪657387....
..                                     ...  ...                              ␣
↪  ...
72  POINT Z (344013.9924472624 675799....  ...  POINT Z (343624.6718084301␣
↪675485....
73  POINT Z (343206.4015371799 675215....  ...  POINT Z (342724.5813225449␣
↪675084....
74  POINT Z (342238.3969935408 674967....  ...  POINT Z (341752.5156862524␣
↪674849....
75  POINT Z (341298.8661618733 674644....  ...  POINT Z (340879.879799073␣
↪674373.2...
76  POINT Z (340414.7337679755 674190....  ...  POINT Z (340274.1556065303␣
↪674138....
[77 rows x 3 columns]
>>> nodes.columns.to_list()
['subsect_a', 'subsect_b', 'subsect_centroid']
```

FeatureCollator.integrate_data

FeatureCollator.**integrate_data**(*element*, *direction*, *subsect_len*, *elr=None*, *ret_dat=False*, *verbose=False*, ***kwargs*)

Construct an integrated data set for the development of a machine learning model for track fixity.

Parameters

- **element** (*str*) – Element of rail head, such as left or right top of rail or running edge.
- **direction** (*str*) – Railway direction, such as up or down direction.
- **subsect_len** (*int*) – Length (in metre) of a subsection for which movement are calculated; defaults to 10.

- **elr** (*str | list | None*) – Engineer's Line Reference(s); defaults to None.
- **ret_dat** (*bool*) – Whether to return the created data set; defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to False.

Returns

An integrated data set for the development of a machine learning model for track fixity.

Return type

pandas.DataFrame

Examples:

```
>>> from src.shaft import FeatureCollator
>>> fc = FeatureCollator(elr=['ECM7', 'ECM8'])
Initialising the feature collator ... Done.
>>> fc.integrate_data(
...     element='Left Top', direction='Up', subsect_len=10, verbose=True)
>>> fc.track_movement_
                                            subsection  ...  Stations
0     LINESTRING Z (399428.96 653473.9 34.442, 39942...  ...         0
1     LINESTRING Z (399434.5201166851 653482.2116640...  ...         0
2     LINESTRING Z (399440.0608295194 653490.5362699...  ...         0
3     LINESTRING Z (399445.5700898784 653498.8817273...  ...         0
4     LINESTRING Z (399451.0456262118 653507.2493444...  ...         0
                                                    ...  ...       ...
7625  LINESTRING Z (340180.451306418 674103.12674018...  ...         0
7626  LINESTRING Z (340171.0809418422 674099.6346047...  ...         0
7627  LINESTRING Z (340161.7096778383 674096.1448800...  ...         0
7628  LINESTRING Z (340152.3387371677 674092.6542742...  ...         0
7629  LINESTRING Z (340142.9690238989 674089.1603813...  ...         0
[7630 rows x 46 columns]
>>> fc.track_movement_.columns.to_list()[-10:]
['vertical_displacement_b_abs_max',
 'Curvature',
 'Cant',
 'Max speed',
 'Max axle load',
 'Overline bridges',
 'Retaining walls',
 'Tunnels',
 'Underline bridges',
 'Stations']
```

FeatureCollator.make_mileage_sequence

static FeatureCollator.**make_mileage_sequence**(*dat*, *start_mil_col*, *end_mil_col=None*)

Put together all values of the mileages into a sequence.

Parameters

- **dat** (*pandas.DataFrame*) – A data set in which mileage data (of both start and end) is available.
- **start_mil_col** (*str*) – Column name of the start mileage data.
- **end_mil_col** (*str | None*) – Column name of the end mileage data; defaults to None.

Returns

A sequence of mileages in the given data set dat, and distances (in metres) between adjacent mileages.

Return type

pandas.DataFrame

Examples:

```
>>> from src.shaft import FeatureCollator
>>> test_elr = 'ECM8'
>>> fc = FeatureCollator(elr=test_elr)
Initialising the feature collator ... Done.
>>> ballast_summary = fc.ballast.load_data(elr=test_elr)
>>> mil_seq = fc.make_mileage_sequence(ballast_summary, 'StartMileage')
>>> mil_seq
      Mileage  metre_to_next
0      0.0618         9.1440
1      0.0628         1.8288
2      0.0630        17.3736
3      0.0649         8.2296
4      0.0658         1.8288
       ...            ...
2851  54.1100         0.9144
2852  54.1101         0.9144
2853  54.1102         1.8288
2854  54.1104         2.7432
2855  54.1107         0.0000
[2856 rows x 2 columns]
>>> mil_seq = fc.make_mileage_sequence(ballast_summary, 'StartMileage',
↪'EndMileage')
>>> mil_seq
      Mileage  metre_to_next
0      0.0618         9.1440
1      0.0628         1.8288
2      0.0630        17.3736
3      0.0649         8.2296
4      0.0658         1.8288
       ...            ...
2852  54.1101         0.9144
```

(continues on next page)

(continued from previous page)

```
2853  54.1102         1.8288
2854  54.1104         2.7432
2855  54.1107         0.9144
2856  54.1108         0.0000
[2857 rows x 2 columns]
```

FeatureCollator.make_subsection_buffer

FeatureCollator.**make_subsection_buffer**(*subsect_geom*, *buf_type=1*, *buf_dist=None*, *ret_dat=True*)

Make a buffer for every subsection for which track movement is calculated.

Parameters

- **subsect_geom** (*pandas.Series | list | numpy.ndarray | LineString*) – A sequence of geometry objects each representing a track subsection
- **buf_type** (*int*) – Buffer type; defaults to 1; options include 1 (circle), 2 (flat) and 3 (square); see shapely.geometry.CAP_STYLE for more details.
- **buf_dist** (*int | float | None*) – Radius of the buffer; defaults to None; see shapely.geometry.buffer for more details.
- **ret_dat** (*bool*) – Whether to return the processed data; defaults to True.

Returns

Buffer(s) for track subsection(s) for which track movement is calculated.

Return type

pandas.Series | list | numpy.ndarray | Polygon

Examples:

```
>>> from src.shaft import FeatureCollator
>>> fc = FeatureCollator(elr='ECM8')
Initialising the feature collator ... Done.
>>> trk_movement = fc.load_movement(
...     element='Left Top', direction='Up', subsect_len=1000)
>>> trk_movement_ = list(trk_movement.values())[0]
>>> sub_geom = trk_movement_.subsection
>>> sub_geom
0     LINESTRING Z (399428.96 653473.9 34.442, 39942...
1     LINESTRING Z (399611.8618926046 654427.6398435...
2     LINESTRING Z (399170.8610524134 655321.5851676...
3     LINESTRING Z (398643.3364543988 656169.1074680...
4     LINESTRING Z (397987.7488072964 656909.7001183...
                            ...
72    LINESTRING Z (344013.9924472624 675799.4646636...
73    LINESTRING Z (343206.4015371799 675215.9293591...
```

(continues on next page)

(continued from previous page)

```
74    LINESTRING Z (342238.3969935408 674967.5756095...
75    LINESTRING Z (341298.8661618733 674644.9047881...
76    LINESTRING Z (340414.7337679755 674190.3677420...
Name: subsection, Length: 77, dtype: object
>>> sub_buffers = fc.make_subsection_buffer(sub_geom, buf_type=1)
>>> sub_buffers
0     POLYGON ((399641.74128885 653931.2746024097, 3...
1     POLYGON ((399432.8613179412 654884.0547386116,...
2     POLYGON ((398947.8690315923 655758.5577137272,...
3     POLYGON ((398308.7644842083 656519.5781464423,...
4     POLYGON ((397864.7254107677 657387.039114261, ...
                            ...
72    POLYGON ((343644.6718084301 675485.7349224968,...
73    POLYGON ((342744.5813225449 675084.2876887596,...
74    POLYGON ((341772.5156862524 674849.6516215525,...
75    POLYGON ((340899.879799073 674373.2162596786, ...
76    POLYGON ((340294.1556065303 674138.0464607034,...
Name: subsection, Length: 77, dtype: object
```

Illustration:

```
import matplotlib.pyplot as plt
import numpy as np
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg', font_name='Times New Roman')

fig = plt.figure(figsize=(11, 5))
ax = fig.add_subplot(aspect='equal', adjustable='box')

colours = plt.get_cmap('tab10').colors

for g in sub_geom:
    g = np.array(g.coords)
    ax.plot(g[:, 0], g[:, 1], color=colours[0])
plt.plot([], [], color=colours[0], label='Track subsections')

for buf in sub_buffers:
    buf = np.array(buf.exterior.coords)
    ax.plot(buf[:, 0], buf[:, 1], color=colours[1])
plt.scatter([], [], marker='o', fc='none', ec=colours[1], label='Buffers')

ax.set_xlabel('Easting', fontsize=14, labelpad=8.0)
ax.set_ylabel('Northing', fontsize=14, labelpad=8.0)

ax.legend()
fig.tight_layout()

fig.show()

# from pyhelpers.store import save_figure
# fig_filename = "fc_make_subsection_buffer_demo"
# save_figure(fig, f"docs\source\_images\{fig_filename}.svg", verbose=True)
# save_figure(fig, f"docs\source\_images\{fig_filename}.pdf", verbose=True)

plt.close(fig)
```

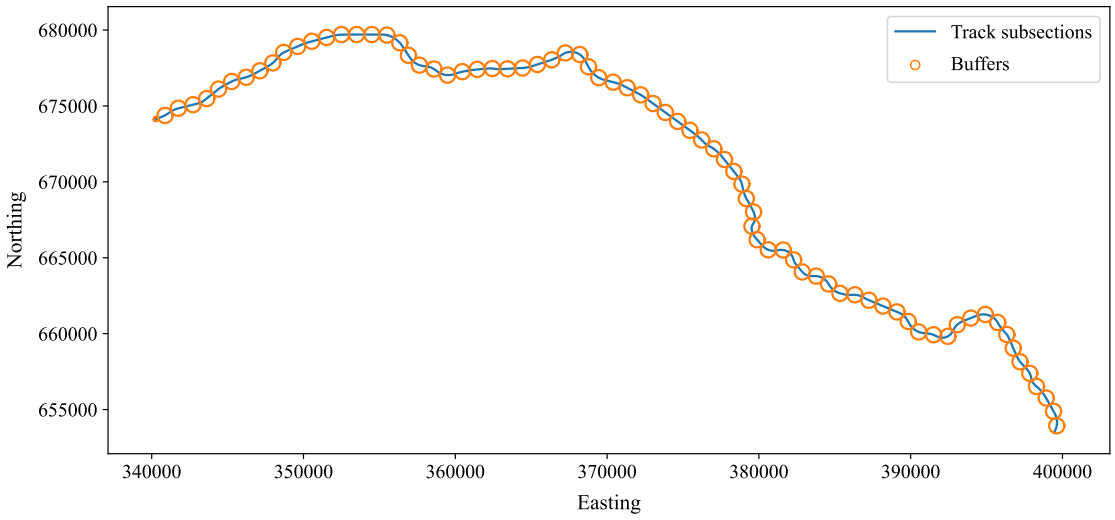


Figure 97: Buffers for track subsections (for which track movement is calculated).

FeatureCollator.view_subsection_buffer

FeatureCollator.**view_subsection_buffer**(*buf_type=1*, *buf_dist=None*, *fig_size=(11, 5)*, *save_as=None*, *dpi=600*, ***kwargs*)

View subsection buffer.

Parameters

- **buf_type** (*int*) – Buffer type; defaults to 1; options include 1 (circle), 2 (flat) and 3 (square); see shapely.geometry.CAP_STYLE for more details.
- **buf_dist** (*int | float | None*) – Radius of the buffer; defaults to None.; see shapely.geometry.buffer for more details.
- **fig_size** (*tupule*) – Figure size; defaults to (11, 5).
- **save_as** (*str | None*) – File format that the view is saved as; defaults to None.
- **dpi** (*int | None*) – DPI for saving image; defaults to 600.
- **kwargs** – [Optional] parameters of the method *TrackMovement.load_movement()*.

Examples:

```
>>> from src.shaft import FeatureCollator
>>> from pyhelpers.settings import mpl_preferences
>>> fc = FeatureCollator(elr='ECM8')
Initialising the feature collator ... Done.
>>> # mpl_preferences(backend='TkAgg')
>>> # fc.view_subsection_buffer(
... #     element='Left Top', direction='Up', subsect_len=1000, save_as=".svg")
```

(continues on next page)

(continued from previous page)

```
>>> fc.view_subsection_buffer(element='Left Top', direction='Up', subsect_
↪len=1000)
```

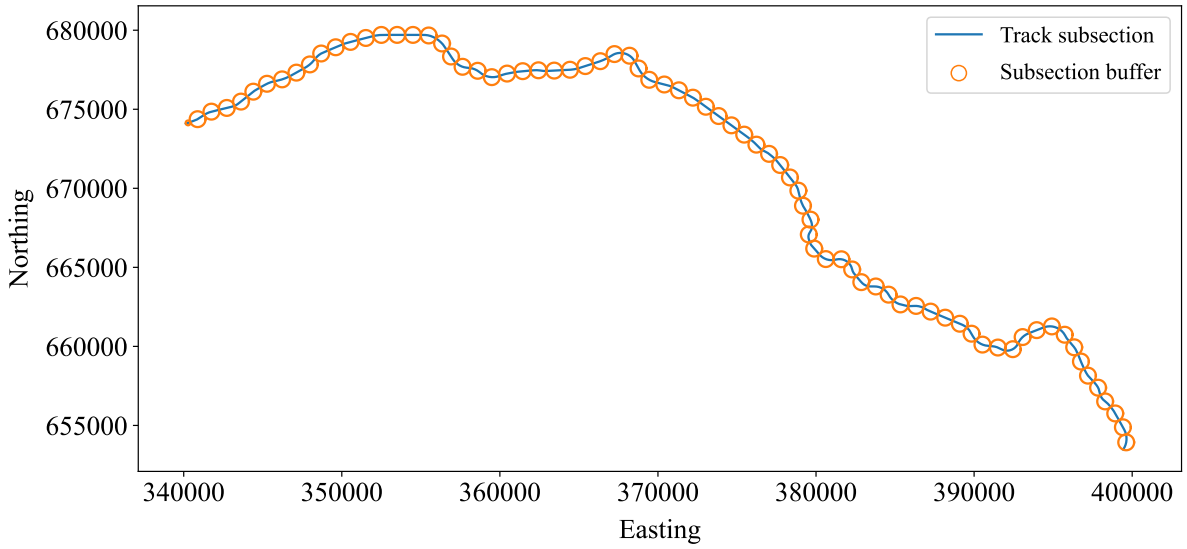


Figure 98: Buffers for ECM8 track subsections of ~1km.

2.3.2 Utilities

This following utilities are used specifically for facilitating the calculation of track movement.

find_closest_subsections(cen1, cen2, sect_base)	Find the closest subsections.
weld_subsections(krdz_dat, start[, as_geom])	Weld a number of track subsections together into a single section.
split_section(section[, unit_length, ...])	Split a track section into a number of subsections, with each having approximately equal lengths.
rearrange_line_points(ls1, ls2[, ref, side])	Rearrange points of the given lines to adjust the direction of one to the other.
calculate_unit_subsection_displacement()	Calculate the displacement of a (short) section (of ~one metre) between two different times.
calc_unit_subsec_disp(unit_sections, ref_obj)	Calculate the lateral, vertical, and cartesian displacements between two unit sections.
get_linestring_ends(ls)	Get the two ends of a line.
sort_line_points(line_geoms[, as_geom])	Sort a series of LineStrings of the tile-based classified KRDZ data to form a line.

find_closest_subsections

src.shaft.sec_utils.**find_closest_subsections**(*cen1*, *cen2*, *sect_base*)

Find the closest subsections.

Parameters

- **cen1** (*numpy.ndarray*) – Centroid of one track subsection.
- **cen2** (*numpy.ndarray*) – Centroid of another track subsection.
- **sect_base** (*shapely.geometry.base.GeometrySequence*) – The reference object. (In this project, it is the relatively longer track subsection).

Returns

The closest subsection given the shortest distance between `cen1` and `cen2`, taking `sect_base` as the reference object.

Return type

shapely.geometry.base.GeometrySequence

weld_subsections

src.shaft.sec_utils.**weld_subsections**(*krdz_dat*, *start*, *as_geom=True*)

Weld a number of track subsections together into a single section.

Parameters

- **krdz_dat** (*pandas.DataFrame*) – KRDZ data loaded from the project database.
- **start** (*tuple | list | Point*) – Point at which the polyline (i.e. welded track section) starts.
- **as_geom** (*bool*) – Whether to return a geometry object; when `as_geom=False`, the method returns an array of coordinates; defaults to `True`.

Returns

A track section consisting of several subsections.

Return type

LineString | numpy.ndarray

Examples:

```
>>> from src.shaft.sec_utils import weld_subsections
>>> from src.shaft import KRDZGear
>>> krdzg = KRDZGear()
>>> tiles_xys = [(360100, 677100), (360200, 677100), (360000, 677100)]
>>> # Top of the right rail in the down direction (within the above tiles) in April␣
 ↪2020
>>> krdz_dat_202004_drt = krdzg.load_classified_krdz(
...     tile_xy=tiles_xys, pcd_date='202004', direction='down', element='right top')
```

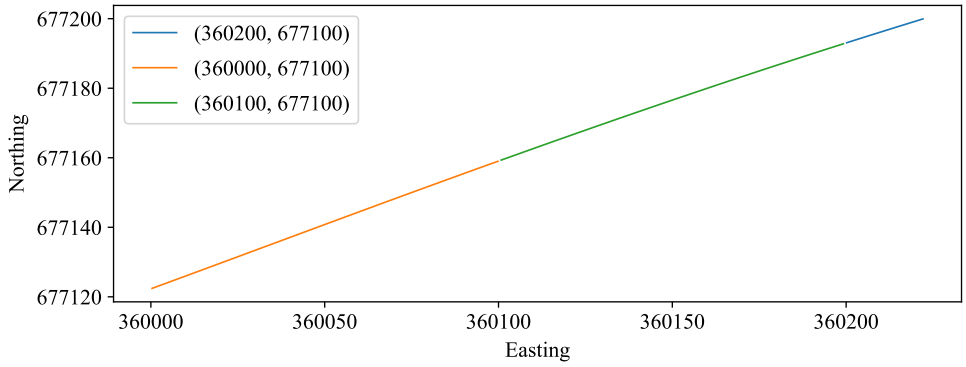
(continues on next page)

(continued from previous page)

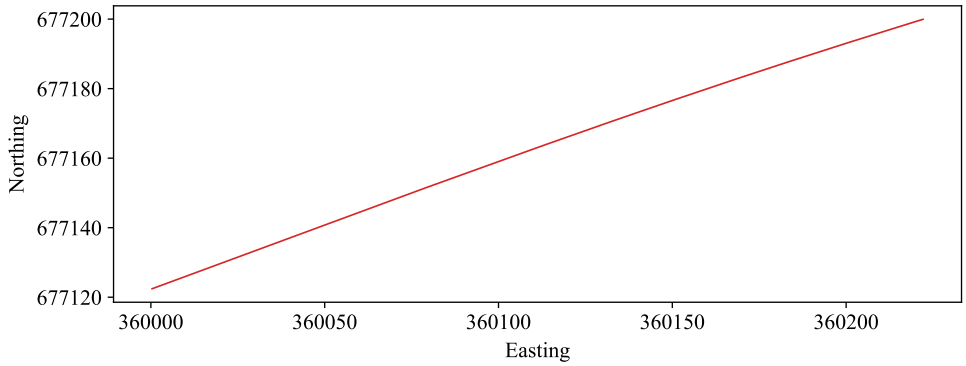
```
>>> krdz_dat_202004_drt
   Year  Month  Tile_X  Tile_Y Direction         Element                       ␣
 ↪geometry
0  2020      4  360200  677100      Down  RightTopOfRail  LINESTRING Z (360200.16 6..
 ↪.
1  2020      4  360000  677100      Down  RightTopOfRail  LINESTRING Z (360000.349 ..
 ↪.
2  2020      4  360100  677100      Down  RightTopOfRail  LINESTRING Z (360100.773 ..
 ↪.
>>> # Sum of the lengths of the three subsections
>>> krdz_dat_202004_drt['geometry'].map(lambda x: x.length).sum()
232.95245581071197
>>> drt_202004_welded = weld_subsections(krdz_dat_202004_drt, start=(360000, 677100))
>>> # Total length (where the additional ~2 metres are from the two "joints")
>>> drt_202004_welded.length
234.9511338346051
```

Illustration:

```
import matplotlib.pyplot as plt
import matplotlib.gridspec as gs
from pyhelpers.settings import mpl_preferences

mpl_preferences(backend='TkAgg', font_name='Times New Roman')

fig = plt.figure(figsize=(8, 7), constrained_layout=True)
mgs = gs.GridSpec(2, 1, figure=fig)

colours = plt.get_cmap('tab10').colors

# Original separate subsections
ax1 = fig.add_subplot(mgs[0, :], aspect='equal', adjustable='box')
for i in krdz_dat_202004_drt.index:
    ls = krdz_dat_202004_drt['geometry'][i]
    ls_xs, ls_ys = ls.coords.xy
    tile_xy = tuple(krdz_dat_202004_drt.loc[i, ['Tile_X', 'Tile_Y']].astype(int).
 ↪tolist())
    ax1.plot(ls_xs, ls_ys, linewidth=1, color=colours[i], label=f'{tile_xy}')
ax1.set_xlabel('Easting', fontsize=13, labelpad=5)
ax1.set_ylabel('Northing', fontsize=13, labelpad=5)
ax1.set_title('(a) Original subsections', y=0, pad=-55)
ax1.legend()

# Welded section
ax2 = fig.add_subplot(mgs[1, :], aspect='equal', adjustable='box')
welded_xs, welded_ys = drt_202004_welded.coords.xy
ax2.plot(welded_xs, welded_ys, linewidth=1, color=colours[i + 1])
ax2.set_xlabel('Easting', fontsize=13, labelpad=5)
ax2.set_ylabel('Northing', fontsize=13, labelpad=5)
ax2.set_title('(b) Welded section', y=0, pad=-55)

# from pyhelpers.store import save_figure
# fig_pathname = "docs/source/_images/tm_weld_subsections_demo"
# save_figure(fig, f"{fig_pathname}.svg", verbose=True)
# save_figure(fig, f"{fig_pathname}.pdf", verbose=True)
```



(a) Original subsections



(b) Welded section

Figure 99: Welded section of the top of the right rail of in the down direction within the tiles (360000, 677100), (360100, 677100) and (360200, 677100).

split_section

src.shaft.sec_utils.**split_section**(*section*, *unit_length=1*, *coarsely=False*, *use_original_points=True*, *to_geoms=False*)

Split a track section into a number of subsections, with each having approximately equal lengths.

Parameters

- **section** (*numpy.ndarray | LineString*) – Track section (represented as a polyline).
- **unit_length** (*int | float*) – Length of each subsection; defaults to 1 (in metre).
- **coarsely** (*bool*) – Whether to split the section in a coarse way; defaults to False.
- **use_original_points** (*bool*) – Whether to use original points as splitters; defaults to False.
- **to_geoms** (*bool*) – Whether to transform the obtained geometry object (GeometryCollection) to its iterable form (GeometrySequence); defaults to True.

Returns
A sequence of approximately equal-length subsections of the polyline object.

Return type
list | GeometryCollection

See also:

- Examples for the method `TrackMovement.split_section()`.

rearrange_line_points

`src.shaft.sec_utils.`**`rearrange_line_points`**(*ls1*, *ls2*, *ref=None*, *side='left'*)

Rearrange points of the given lines to adjust the direction of one to the other.

Parameters

- **ls1** (*LineString*) – One line.
- **ls2** (*LineString*) – Another line.
- **ref** (*LineString | None*) – A reference line by which the direction of the given lines are determined; when `ref=None` (default), the function uses `ls1` as the reference.
- **side** (*str*) – The `side` parameter used by shapely.geometry.LineString.parallel_offset(); defaults to `'left'`.

Returns
Two lines with rearranged points against each other.

Return type
tuple

See also:

- Examples for the method `calculate_section_movement()`.

calculate_unit_subsection_displacement

`src.shaft.sec_utils.`**`calculate_unit_subsection_displacement`**(*unit_sect_orig*, *unit_sect_shifted*, *ref_obj*)

Calculate the displacement of a (short) section (of ~one metre) between two different times.

Parameters

- **unit_sect_orig** (*numpy.ndarray | LineString*) – A unit track section's position at an earlier time.

- **unit_sect_shifted** (*numpy.ndarray | LineString*) – (Almost) the same unit track section' position at a later time.
- **ref_obj** (*numpy.ndarray | LineString*) – A reference object for identifying direction of the displacement.

Returns
Both lateral and vertical displacements.

Return type
tuple

calc_unit_subsec_disp

src.shaft.sec_utils.**calc_unit_subsec_disp**(*unit_sections*, *ref_obj*)

Calculate the lateral, vertical, and cartesian displacements between two unit sections.

Parameters

- **unit_sections** (*tuple*) – A tuple containing the original and shifted unit sections.
- **ref_obj** (*numpy.ndarray | LineString*) – Reference object used for displacement calculations.

Returns
A list containing lateral displacement, vertical displacement, and cartesian distance.

Return type
list

get_linestring_ends

src.shaft.sec_utils.**get_linestring_ends**(*ls*)

Get the two ends of a line.

Parameters
ls (*LineString*) – A line.

Returns
Two points representing the two ends of ls.

Return type
tuple

sort_line_points

src.shaft.sec_utils.**sort_line_points**(*line_geoms*, *as_geom=True*)

Sort a series of LineStrings of the tile-based classified KRDZ data to form a line.

Parameters

- **line_geoms** (*pandas.Series*) – A series of LineStrings of the tile-based classified KRDZ data.
- **as_geom** (*bool*) – Whether to return as LineString type; when as_geom=False, the method returns an array; defaults to True.

Returns

An array of continuous line points or an object of LineString.

Return type

np.ndarray | LineString

2.4 modeller

Applying machine learning algorithms to data generated from *shaft* to create a model for track movement prediction.

This module provides functionalities to develop and evaluate machine learning models that predict track movement based on data generated by the *shaft* module. It includes classes for further processing data, training machine learning models, evaluating model performance, and visualising results.

See also:

- Documentation of the *shaft* module.
- Example usage in the method classifier().

Classes

TrackMovementEstimator(element, direction[, ...])	Prototype machine-learning-based probabilistic model for predicting track movement.

2.4.1 TrackMovementEstimator

class src.modeller.**TrackMovementEstimator**(*element*, *direction*, *subsect_len=10*, *elr=None*, *target_name=None*, *feature_names=None*, *db_instance=None*, *random_state=0*, *verbose=True*)

Prototype machine-learning-based probabilistic model for predicting track movement.

This class implements a prototype model designed to predict track movement and analyse track movement using the random forest algorithm. The model is trained and tested on a data set processed by the *shaft* module, where the *TrackMovement* class calculates the track movement (serving as the target variable in the model) and the *FeatureCollator* class gathers information of relevant explanatory variables (forming the feature data set for the model).

Parameters

- **element** (*str*) – Element of rail head, such as left or right top of rail, or running edge.
- **direction** (*str*) – Railway direction, such as up or down direction.
- **subsect_len** (*int*) – Length (in metres) of a subsection for which movement is calculated; defaults to 10.
- **elr** (*str | list | None*) – Engineer's Line Reference(s); defaults to None.
- **target_name** (*str | None*) – Name of the target variable; when target_name=None (default), it takes the class attribute TARGET_NAME.
- **feature_names** (*list | None*) – Names of explanatory variables (features) for model specification; defaults to None.
- **db_instance** (TrackFixityDB | *None*) – PostgreSQL database instance; defaults to None.
- **random_state** (*int | None*) – A random seed number; defaults to 0.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to True.

Variables

- **data_set** (*pandas.DataFrame | None*) – The data set for developing the prototype machine learning model for estimating/predicting track movement; defaults to None.
- **valid_target_names** (*list*) – List of valid target variable names.
- **target_name** (*str*) – The name of the target variable.

- **target_edges** (*numpy.ndarray | None*) – Critical values based on which the target data is categorised into different classes; defaults to None.
- **label_encoder** (*LabelEncoder*) – An encoder that labels the target data.
- **target_data** (*pandas.Series*) – Data of track movement to be predicted; defaults to None.
- **target_data_labels** (*pandas.Series*) – Labelled target_data; defaults to None.
- **labels** (*list*) – Descriptive texts for target_data_labels; defaults to None.
- **feature_names** (*list*) – Names of explanatory variables (features) for model specification.
- **feature_data** (*pandas.DataFrame*) – Data of features collated to predict the target_data_labels.
- **random_state** (*int | None*) – A random seed number.
- **X_train** (*pandas.DataFrame | None*) – Feature data of a training set; defaults to None.
- **y_train** (*pandas.DataFrame | None*) – Target data of a training set; defaults to None.
- **X_test** (*pandas.Series | None*) – Feature data of a test set; defaults to None.
- **y_test** (*pandas.Series | None*) – Target data of a test set; defaults to None.
- **estimator** (*RandomForestClassifier*) – The prototype random forest model; defaults to None.
- **best_estimator** (*RandomForestClassifier*) – The best random forest model (if a model selection procedure is implemented); defaults to None.
- **best_estimator_params** (*pandas.DataFrame*) – The parameters of the best random forest model; defaults to None.
- **score** (*float*) – The mean accuracy of the estimator making predictions on a given test data; defaults to None.
- **feature_importance** (*pandas.DataFrame*) – Relative importance of different features based on a trained model; defaults to None.
- **confusion_matrix** (*pandas.DataFrame*) – Relative importance of different features based on a trained model; defaults to None.

Examples:

```
>>> from src.modeller.prototype import TrackMovementEstimator
>>> tme = TrackMovementEstimator(element='Left Top', direction='Up', subsect_len=10)
Initialising the feature collator ... Done.
Initialising the estimator ... Done.
>>> tme.target_name
'lateral_displacement_mean'
>>> tme.feature_names
['Curvature',
 'Cant',
 'Max speed',
 'Max axle load',
 'Overline bridges',
 'Underline bridges',
 'Retaining walls',
 'Tunnels',
 'Stations']
```

Attributes:

FEATURE_NAMES	The names of explanatory variables (a.k.a.
TARGET_NAME	The name of the target variable.

TrackMovementEstimator.FEATURE_NAMES

TrackMovementEstimator.**FEATURE_NAMES: list = ['Curvature', 'Cant', 'Max speed', 'Max axle load', 'Overline bridges', 'Underline bridges', 'Retaining walls', 'Tunnels', 'Stations']**

The names of explanatory variables (a.k.a. features) for model specification.

TrackMovementEstimator.TARGET_NAME

TrackMovementEstimator.**TARGET_NAME: str = 'lateral_displacement_mean'**

The name of the target variable.

Methods:

classifier([random_state, verbose, n_jobs])	Train a random forest model for estimating/predicting track movement.
get_descriptive_labels([ret_dat])	Retrieve descriptive texts for the numeric target labels.
get_training_test_sets([random_state, ...])	Create and return training and test data sets.
label_target(target_data[, percentiles, ...])	Categorise the model target (e.g. lateral displacement of the top of a rail) into classes.
view_confusion_matrix([normalise, cmap, ...])	Create a visual representation of the confusion matrix for the current random forest model.

TrackMovementEstimator.classifier

TrackMovementEstimator.**classifier**(*random_state=None*, *verbose=1*, *n_jobs=0*, ***kwargs*)

Train a random forest model for estimating/predicting track movement.

This method initialises and trains a random forest classifier using the provided parameters.

Parameters

- **random_state** (*int | None*) – A random seed number for reproducibility; defaults to None.
- **verbose** (*int | bool*) – Whether to print relevant information in console, or the level of verbosity for the console output; defaults to 1.
- **n_jobs** (*int*) – Number of CPU cores to use for training the classifier. When n_jobs=0 (default), it uses one less than the total number of available cores.
- **kwargs** – [Optional] additional parameters for the sklearn.ensemble.RandomForestClassifier class.

Examples:

```
>>> from modeller import TrackMovementEstimator
>>> element = 'Left Top'
>>> direction = 'Up'
>>> subsect_len = 10
>>> tme = TrackMovementEstimator(element, direction, subsect_len)
Initialising the feature collator ... Done.
Initialising the estimator ... Done.
>>> tme.integrate_data(element, direction, subsect_len, verbose=True)
Calculating track movement ... Done.
```

(continues on next page)

(continued from previous page)

```
Collating features:
  ballasts (src. Ballast summary) ... Done.
  structures (src. CARRS and OPAS) ... Done.
Finalising the data integration ... Done.
>>> list(tme.track_movement.keys())
['Up_LeftTopOfRail_201910_202004']
>>> tme.track_movement['Up_LeftTopOfRail_201910_202004'].head()
                                     subsection  ...  vertical_displacement_b_
↪abs_max
0     LINESTRING Z (399428.96 653473.9 34.442...  ...                        -0.
↪003203
1     LINESTRING Z (399434.5201166851 653482....  ...                        -0.
↪002794
2     LINESTRING Z (399440.0608295194 653490....  ...                        -0.
↪003202
3     LINESTRING Z (399445.5700898784 653498....  ...                        -0.
↪001803
4     LINESTRING Z (399451.0456262118 653507....  ...                        -0.
↪001201
[5 rows x 37 columns]
>>> tme.get_training_test_sets(test_size=0.2, random_state=1)
>>> tme.classifier(n_estimators=300, max_depth=15, oob_score=True)
Mean accuracy: 49.96%

                   Importance
Curvature              0.3887
Cant                   0.3725
Max speed              0.2016
Underline bridges      0.0095
Overline bridges       0.0073
Max axle load          0.0067
Retaining walls        0.0060
Tunnels                0.0058
Stations               0.0019
>>> from pyhelpers.settings import mpl_preferences
>>> mpl_preferences(backend='TkAgg')
>>> # tme.view_confusion_matrix(save_as=".svg", verbose=True)
>>> tme.view_confusion_matrix()
```

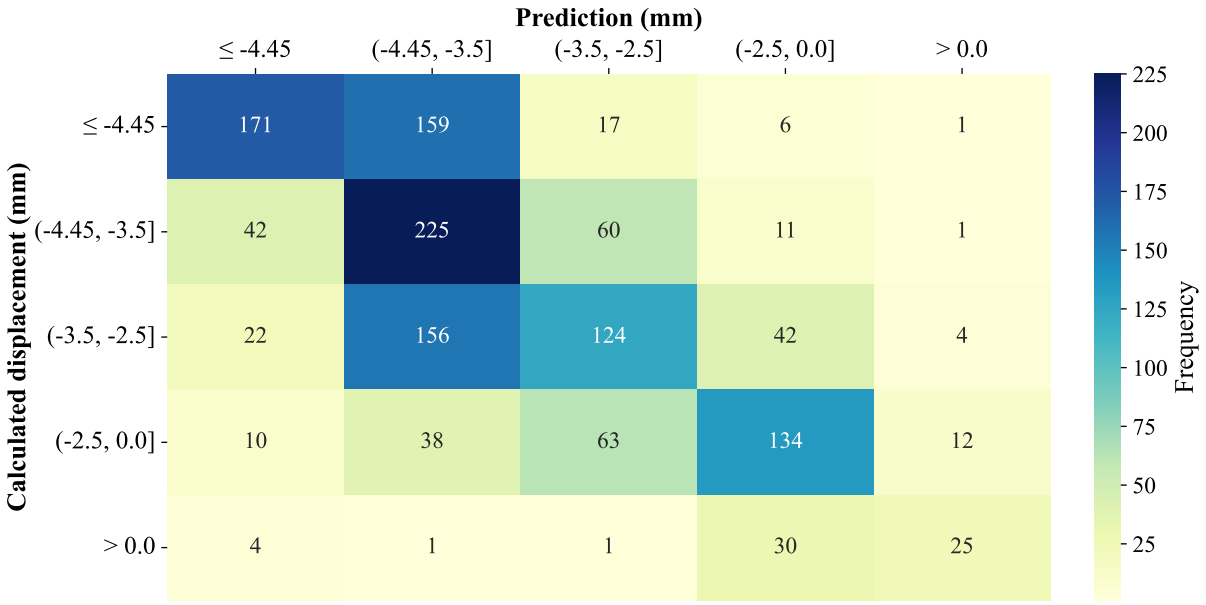


Figure 100: A view of the confusion matrix for the example random forest model.

```
>>> # Normalise the confusion matrix over the predicted labels
>>> # tme.view_confusion_matrix(normalise='pred', save_as=".svg", verbose=True)
>>> tme.view_confusion_matrix(normalise='pred')
```

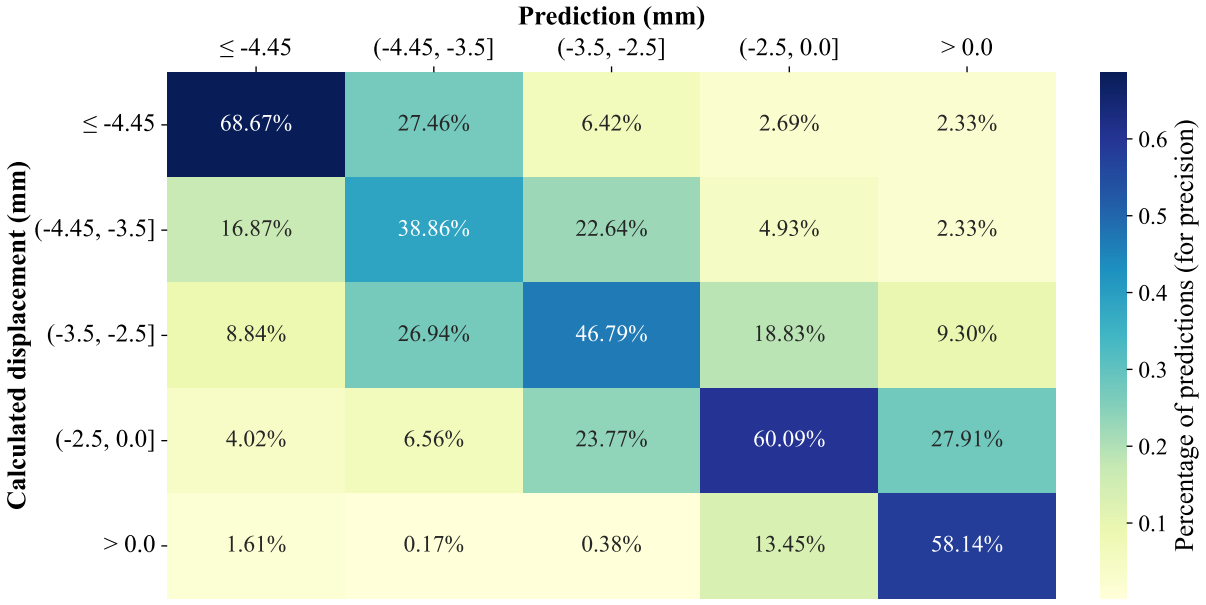


Figure 101: A view of the confusion matrix (normalised over the predicted labels) for the example random forest model.

TrackMovementEstimator.get_descriptive_labels

TrackMovementEstimator.**get_descriptive_labels**(*ret_dat=False*)

Retrieve descriptive texts for the numeric target labels.

This method provides human-readable descriptions for the numeric labels used in the classification of track movement.

Parameters
ret_dat (*bool*) – If True, returns the processed data; defaults to False.

Returns
Descriptive class labels for the calculated track movement, if ret_dat=True.

Return type
list

See also:

- Examples for the method label_target().

TrackMovementEstimator.get_training_test_sets

TrackMovementEstimator.**get_training_test_sets**(*random_state=None*, *ret_dat=False*, *verbose=True*, ***kwargs*)

Create and return training and test data sets.

This method splits the data set into training and test sets for model development and evaluation.

Parameters

- **random_state** (*int | None*) – A random seed number for reproducibility; defaults to None.
- **ret_dat** (*bool*) – If True, returns the created data sets; defaults to False.
- **verbose** (*bool | int*) – Whether to print relevant information in console; defaults to True.
- **kwargs** – [Optional] additional parameters for the sklearn.model_selection.train_test_split function.

Returns
Feature and target data of both the training and test set.

Return type
tuple[pandas.DataFrame, pandas.Series, pandas.DataFrame, pandas.Series]

Examples:

```
>>> from src.modeller import TrackMovementEstimator
>>> element = 'Left Top'
>>> direction = 'Up'
>>> subsect_len = 10
>>> tme = TrackMovementEstimator(element, direction, subsect_len)
Initialising the feature collator ... Done.
Initialising the estimator ... Done.
>>> tme.get_training_test_sets()
Splitting into training and test sets ... Done.
The data is now ready for modelling.
>>> tme.X_train.head()
      Curvature    Cant  Max speed  ...  Retaining walls  Tunnels  Stations
67    -0.000849  -73.00       75.0  ...                0        0         0
7626   0.000000    0.00      125.0  ...                0        0         0
5554  -0.000675 -140.00      100.0  ...                0        0         0
4226  -0.000163  -42.83      105.0  ...                0        0         0
350    0.000107   13.28       90.0  ...                0        0         0
[5 rows x 9 columns]
>>> tme.y_train.head()
59      3
6788    0
4897    3
3631    3
342     4
Name: lateral_displacement_mean, dtype: int64
>>> tme.X_test.head()
      Curvature    Cant  Max speed  ...  Retaining walls  Tunnels  Stations
5845  -0.000529 -134.48      110.0  ...                0        0         0
178    0.000000    0.00       95.0  ...                0        0         0
5249   0.000000    0.00      110.0  ...                0        0         0
2472  -0.001412 -125.00       75.0  ...                0        0         0
1239   0.001314  134.00       80.0  ...                0        0         0
[5 rows x 9 columns]
>>> tme.y_test.head()
5133    3
170     2
4622    2
2031    2
1082    2
Name: lateral_displacement_mean, dtype: int64
```

TrackMovementEstimator.label_target

TrackMovementEstimator.**label_target**(*target_data*, *percentiles=None*, *as_dataframe=False*, *ret_dat=False*, ***kwargs*)

Categorise the model target (e.g. lateral displacement of the top of a rail) into classes.

This method classifies the target data into six categories for use in a random forest model, which handles categorical data. The target data is divided into ranges based on the specified percentiles and labelled with values from 0 to 5. By default, the boundaries of these intervals are determined by the 10th, 25th, 50th, 75th and 95th percentiles.

Parameters

- **percentiles** (*list[float] | None*) – Percentiles used to calculate the descriptive statistics of `target_data`; defaults to `[.25, .5, .75]` if `None`.
- **target_data** (*pandas.Series*) – The continuous target data to be categorised (e.g. lateral displacement of the top of a rail).
- **as_dataframe** (*bool*) – Whether the labelled target is formatted as a DataFrame; defaults to `False`.
- **ret_dat** (*bool*) – If `True`, returns the processed data; defaults to `False`.
- **kwargs** – [Optional] additional parameters for the pandas.cut() function, excluding `include_lowest`, `duplicates` and `retbins`.

Returns

Encoded target data as a pandas Series or DataFrame.

Return type

pandas.Series | pandas.DataFrame

Examples:

```
>>> from modeller import TrackMovementEstimator
>>> element = 'Left Top'
>>> direction = 'Up'
>>> subsect_len = 10
>>> tme = TrackMovementEstimator(element, direction, subsect_len)
Initialising the feature collator ... Done.
Initialising the estimator ... Done.
>>> tme.integrate_data(element, direction, subsect_len, verbose=True)
Calculating track movement ... Done.
Collating features:
  ballasts (src. Ballast summary) ... Done.
  structures (src. CARRS and OPAS) ... Done.
Finalising the data integration ... Done.
>>> tme.track_movement_.head()
                                          subsection  ...  Stations
0  LINESTRING Z (399428.96 653473.9 34.442, 39942...  ...         0
1  LINESTRING Z (399434.5201166851 653482.2116640...  ...         0
2  LINESTRING Z (399440.0608295194 653490.5362699...  ...         0
3  LINESTRING Z (399445.5700898784 653498.8817273...  ...         0
4  LINESTRING Z (399451.0456262118 653507.2493444...  ...         0
[5 rows x 46 columns]
>>> tme.target_name
'lateral_displacement_mean'
>>> target_data = tme.track_movement_[tme.target_name]
>>> target_data.head()
0   -0.002879
1   -0.002754
2   -0.001841
3   -0.002032
4   -0.003821
Name: lateral_displacement_mean, dtype: float64
```

(continues on next page)

(continued from previous page)

```
>>> tme.label_target(target_data)
>>> tme.target_data_labels.head()
0    2
1    2
2    3
3    3
4    1
Name: lateral_displacement_mean, dtype: int64
>>> tme.get_descriptive_labels()
>>> tme.labels
['≤ -4.48 mm',
 '(-4.48, -3.5] mm',
 '(-3.5, -2.51] mm',
 '(-2.51, 0.0] mm',
 '> 0.0 mm']
```

TrackMovementEstimator.view_confusion_matrix

TrackMovementEstimator.**view_confusion_matrix**(*normalise=None*, *cmap='YlGnBu'*, *add_caption=False*, *save_as=None*, *dpi=600*, *verbose=False*, ***kwargs*)

Create a visual representation of the confusion matrix for the current random forest model.

Parameters

- **normalise** (*str | None*) – Options include `'pred'`, `'true'` and `'all'`:
  - `'pred'`: Normalise the confusion matrix over the predicted labels (columns).
  - `'true'`: Normalise over the true labels (rows).
  - `'all'`: Normalise over the entire population.

  When `normalise=None` (default), the confusion matrix will not be normalised. See also the function sklearn.metrics.confusion_matrix.
- **cmap** (*str*) – Colour map for the heatmap; defaults to `'YlGnBu'`.
- **add_caption** (*bool*) – Whether to add a caption below the figure; defaults to `False`.
- **save_as** (*str | None*) – File format to save the visualisation; defaults to `None`.
- **dpi** (*int*) – DPI (dots per inch) for saving the image; defaults to `600`.
- **verbose** (*bool | int*) – Whether to print relevant information to the console; defaults to `False`.

- **kwargs** – [Optional] additional parameters for the seaborn.heatmap function.

See also:

- Examples for the method `classifier()`.

Chapter 3

Publications

Technical Report

- Fu, Q., Easton, J. M. and Burrow, M. P. N. (2021). Track Fixity Layer. Approved by Network Rail Ltd. [Appendix - Technical Documentation]

Journal Paper

- Fu, Q., Easton, J. M. and Burrow, M. P. N. (2024). Development of an Integrated Computing Platform for Measuring, Predicting, and Analyzing the Profile-Specific Fixity of Railway Tracks. Transportation Research Record, 2678(6), 1-13. doi:10.1177/03611981231191521.

Python Module Index

s

Index

F

G

I

P

R

S

T

U

V

W

Y

Z