fractopo

Release 0.5.3

Nikolas Ovaskainen

May 09, 2023

LINKS

`fractopo` is a Python library/application that contains tools for validating and analysing lineament and fracture trace maps (fracture networks). It is targeted at structural geologists working on the characterization of bedrock fractures from outcrops and through remote sensing. `fractopo` is available as a Python library and through a command-line interface. As a Python library, the use of `fractopo` requires prior (Python) programming knowledge. However, if used through the command-line, using `fractopo` only requires general knowledge of command-line interfaces in your operating system of choice.

- Full Documentation is hosted on Read the Docs

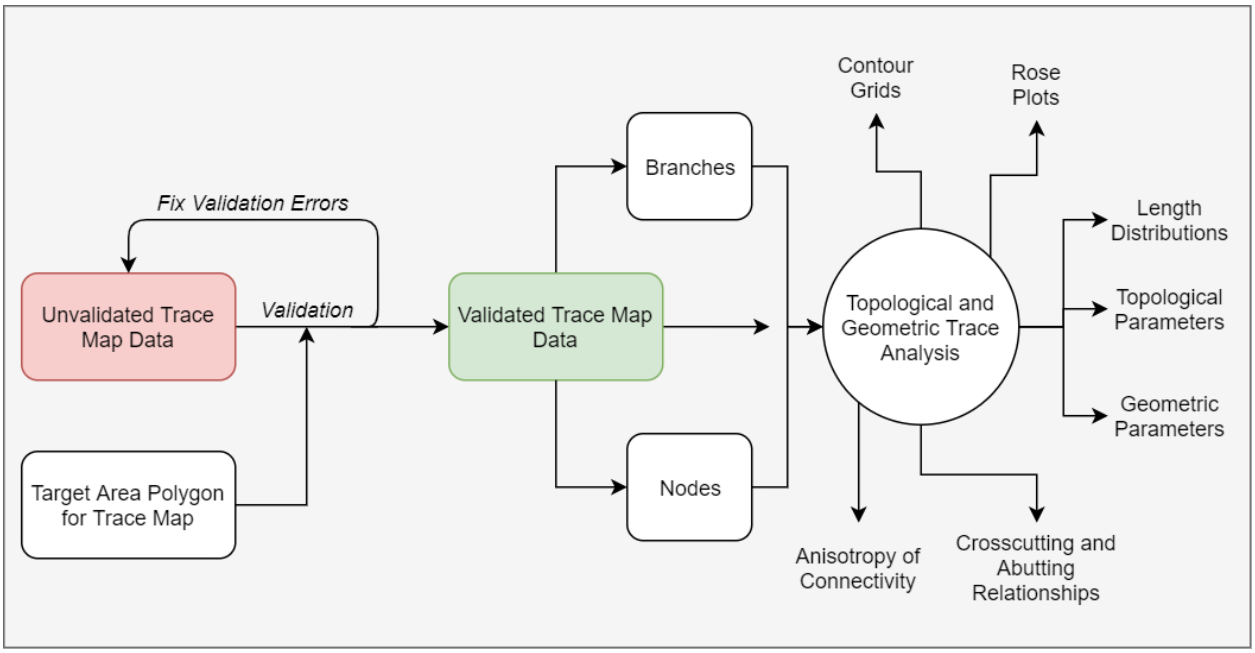


Fig. 1: Overview of fractopo

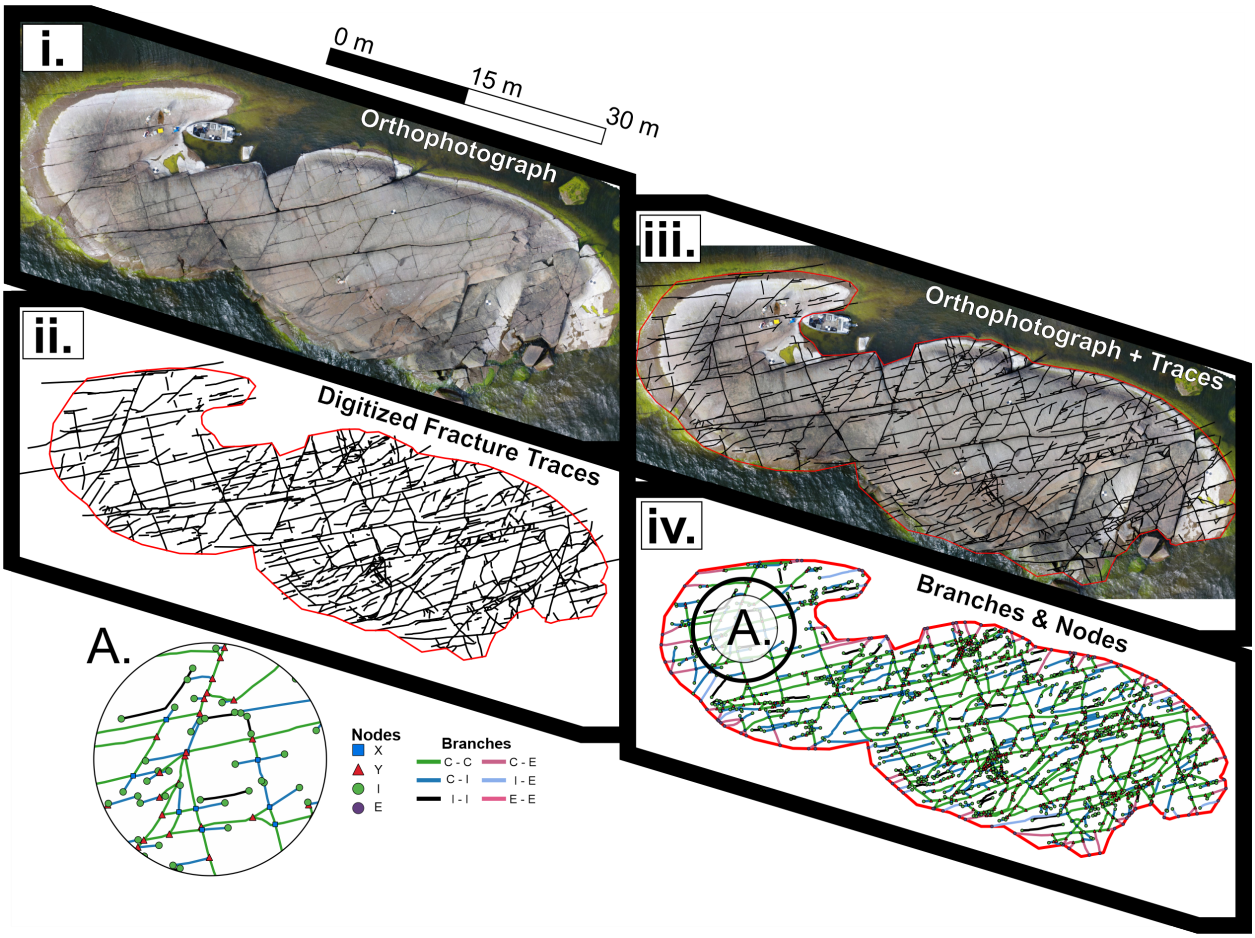


Fig. 2: Visualisation of `fractopo` data. `fractopo` analyses the trace data that can e.g. be digitized from drone orthophotographs (=fractures) or from digital elevation models (=lineaments). The displayed branches and nodes are extracted with `fractopo`.

CHAPTER

ONE

INSTALLATION

`pip` and `poetry` installation only supported for `linux` -based operating systems. For Windows and MacOS install using *(ana)conda*.

1.1 conda

- Only (supported) installation method for `Windows` and `MacOS`!

```
# Create new environment for fractopo (recommended but optional)
conda env create fractopo-env
conda activate fractopo-env
# Available on conda-forge channel
conda install -c conda-forge fractopo
```

1.2 pip

The module is on PyPI.

```
# Non-development installation
pip install fractopo
```

1.3 poetry

For usage:

```
poetry add fractopo
```

For development, only `poetry` installation of `fractopo` is supported:

```
git clone https://github.com/nialov/fractopo
cd fractopo
poetry install
```

CHAPTER

TWO

USAGE

`fractopo` has two main use cases:

1. Validation of lineament & fracture trace data
2. Analysis of lineament & fracture trace data

Validation is done to make sure the data is valid for the analysis and is crucial as analysis cannot take into account different kinds of geometric and topological inconsistencies between the traces. Capabilities and associated guides are inexhaustively listed in the table below.

Functionality	Tutorial/Guide/Example
Validation of trace data	Validation 1; Validation 2
Visualize trace map data	Visualizing
Topological branches and nodes	Network; Topological
Trace and branch length distributions	Length-distributions
Orientation rose plots	Orientation 1; Orientation 2
Plot topological ternary node and branch proportions	Proportions
Cross-cutting and abutting relationships	Relationships 1; Relationships 2;
Geometric and topological fracture network parameters	Parameters
Contour grids of fracture network parameters	Contour-grids
Multi-scale length distributions	Multi-scale

For a short tutorial on use of `fractopo` continue reading:

2.1 Input data

Reading and writing spatial filetypes is done in `geopandas` and you should see `geopandas` documentation for more advanced read-write use cases:

- https://geopandas.org/

Simple example with trace and area data in GeoPackages:

```
import geopandas as gpd

# Trace data is in a file `traces.gpkg` in current working directory
# Area data is in a file `areas.gpkg` in current working directory
trace_data = gpd.read_file("traces.gpkg")
area_data = gpd.read_file("areas.gpkg")
```

2.2 Trace validation

Trace data must be validated using `fractopo` validation functionality before analysis. The topological analysis of lineament & fracture traces implemented in `fractopo` will not tolerate uncertainty related to the topological abutting and snapping relationships between traces. See the documentation for further info on validation error types. Trace validation is recommended before all analysis using `Network`. Trace and target area data can be validated for further analysis with a `Validation` object:

```
from fractopo import Validation

validation = Validation(
    trace_data,
    area_data,
    name="mytraces",
    allow_fix=True,
)

# Validation is done explicitly with `run_validation` method
validated_trace_data = validation.run_validation()
```

Trace validation is also accessible through the `fractopo` command-line interface, `fractopo tracevalidate` which is more straightforward to use than through Python calls. Note that all subcommands of `fractopo` are available by appending them after `fractopo`.

`tracevalidate` always requires the target area that delineates trace data.

```
# Get full up-to-date command-line interface help
fractopo tracevalidate --help

# Basic usage example:
fractopo tracevalidate /path/to/trace_data.shp /path/to/target_area.shp\
   --output /path/to/validated_trace_data.shp

# Or with automatic saving to validated/ directory
fractopo tracevalidate /path/to/trace_data.shp /path/to/target_area.shp\
   --summary
```

2.3 Geometric and topological trace network analysis

`fractopo` can be used to extract lineament & fracture size, abundance and topological parameters from two-dimensional lineament and fracture trace, branch and node data.

Trace and target area data (`GeoDataFrames`) are passed into a `Network` object which has properties and functions for returning and visualizing different parameters and attributes of trace data.

```
from fractopo import Network

# Initialize Network object and determine the topological branches and nodes
network = Network(
    trace_data,
    area_data,
    # Give the Network a name!
```

(continues on next page)

(continued from previous page)

```
    name="mynetwork",
    # Specify whether to determine topological branches and nodes
    # (Required for almost all analysis)
    determine_branches_nodes=True,
    # Specify the snapping distance threshold to define when traces are
    # snapped to each other. The unit is the same as the one in the
    # coordinate system the trace and area data are in.
    # In default values, fractopo assumes a metric unit and using metric units
    # is heavily recommended.
    snap_threshold=0.001,
    # If the target area used in digitization is a circle, the knowledge can
    # be used in some analysis
    circular_target_area=True,
    # Analysis on traces can be done for the full inputted dataset or the
    # traces can be cropped to the target area before analysis (cropping
    # recommended)
    truncate_traces=True,
)

# Properties are easily accessible
# e.g.,
network.branch_counts
network.node_counts

# Plotting is done by plot_ -prefixed methods
network.plot_trace_lengths()
```

Network analysis is also available through the `fractopo` command-line interface but using the Python interface (e.g. `jupyter lab`, `ipython`) is recommended when analysing `Networks` to have access to all available analysis and plotting methods. The command-line entrypoint is **opinionated** in what outputs it produces. Brief example of command-line entrypoint:

```
fractopo network /path/to/trace_data.shp /path/to/area_data.shp\
   --name mynetwork

# Use --help to see all up-to-date arguments and help
fractopo network --help
```

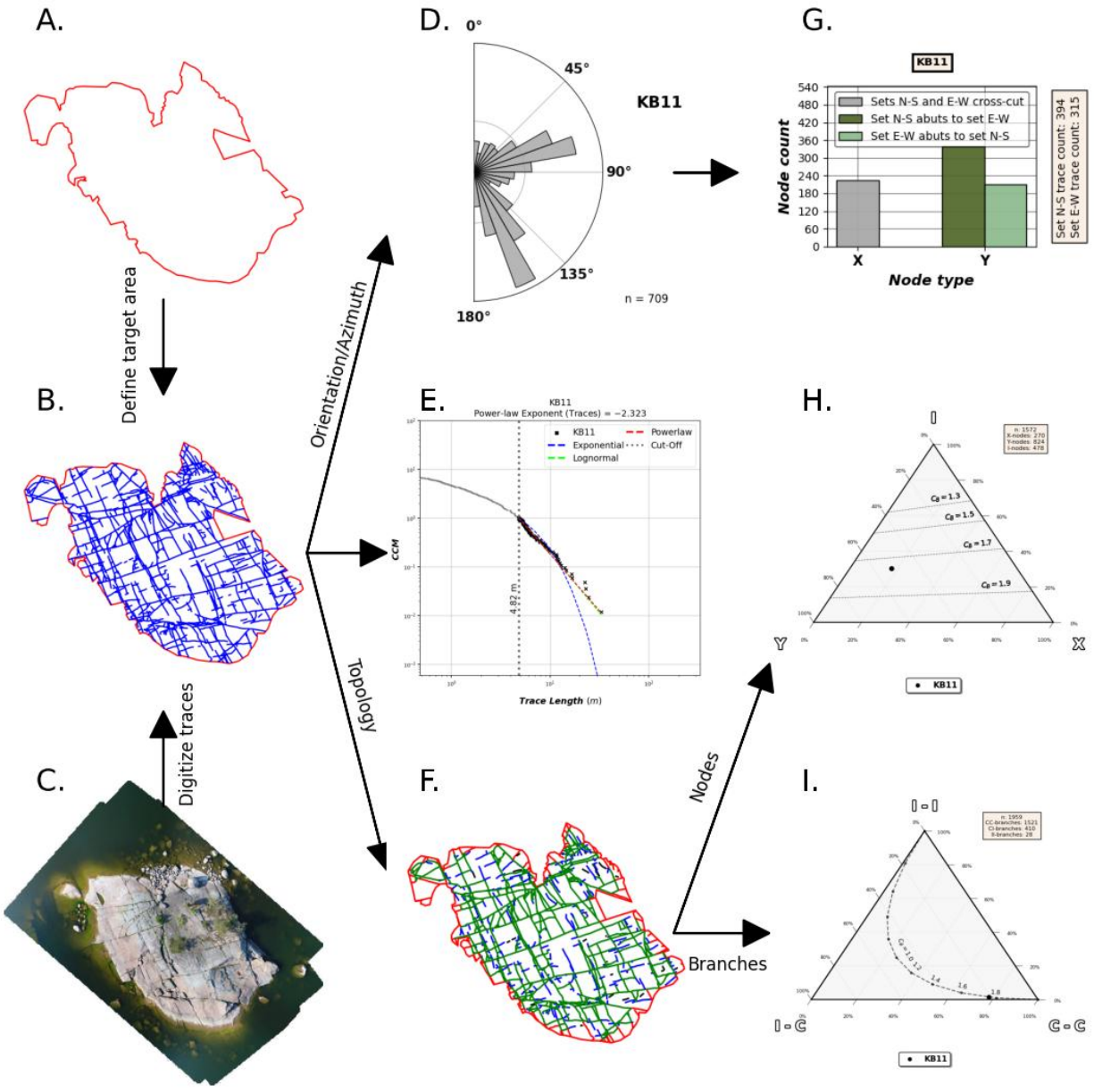


Fig. 1: Data analysis workflow visualisation for fracture trace data (KB11). A. Target area for trace digitisation. B. Digitized traces and target area. C. Orthomosaic used as the base raster from which the traces are digitized from. D. Equal-area length-weighted rose plot of the fracture trace azimuths. E. Length distribution analysis of the trace lengths. F. Determined branches and nodes through topological analysis. G. Cross-cut and abutting relationships between chosen azimuth sets. H. Ternary plot of node (X, Y and I) proportions. I. Ternary plot of branch (C-C, C-I, I-I) proportions.

CHAPTER

THREE

CITING

To cite this software:

- The software is introduced in https://doi.org/10.1016/j.jsg.2022.104528 and you can cite that article as a general citation:

```
Ovaskainen, N., Nordbäck, N., Skyttä, P. and Engström, J., 2022. A new
subsampling methodology to optimize the characterization of
two-dimensional bedrock fracture networks. Journal of Structural Geology,
p.104528.
```

- To cite a specific version of `fractopo` you can use a `zenodo` provided `DOI`. E.g. https://doi.org/10.5281/zenodo.5957206 for version `v0.2.6`. See the `zenodo` page of `fractopo` for the `DOI` of each version: https://doi.org/10.5281/zenodo.5517485

CHAPTER

FOUR

SUPPORT

For issues of any kind: please create a GitHub issue here! Alternatively, you can contact the main developer by email at nikolasovaskainen@gmail.com.

CHAPTER

FIVE

REFERENCES

For the scientific background, prior works, definition of traces, branches and nodes along with the explanation of the plots and the plotted parameters, you are referred to multiple sources:

- Sanderson and Nixon, 2015
  - Trace and branch size, abundance and topological parameter definitions.
- Ovaskainen et al, 2022
  - Application of `fractopo` for subsampling analysis of fracture networks.
- Nyberg et al., 2018
  - A similar package to `fractopo` with a `QGIS` GUI.
  - NetworkGT GitHub
- Sanderson and Peacock, 2020
  - Discussion around rose plots and justification for using length-weighted equal-area rose plots.
- Alstott et al. 2014
  - Length distribution modelling using the Python 3 `powerlaw` package which `fractopo` uses
  - powerlaw GitHub
- Bonnet et al., 2001
  - Length distribution modelling review.
- My Master's Thesis, Ovaskainen, 2020
  - Plots used in my Thesis were done with an older version of the same code used for this plugin.

CHAPTER

SIX

DEVELOPMENT

- The package interfaces are nearing stability and breaking changes in code should for the most part be included in the `CHANGELOG.md` after 25.4.2023. However, this is not guaranteed until the version reaches v1.0.0. The interfaces of `Network` and `Validation` can be expected to be the most stable.
- For general contributing guidelines, see CONTRIBUTING.rst

Development dependencies for `fractopo` include:

- `poetry`
  - Used to handle Python package dependencies.

```
# Use poetry run to execute poetry installed cli tools such as invoke,
# nox and pytest.
poetry run '<cmd>'
```

- `doit`
  - A general task executor that is a replacement for a `Makefile`
  - Understands task dependencies and can run tasks in parallel even while running them in the order determined from dependencies between tasks. E.g., requirements.txt is a requirement for running tests and therefore the task creating requirements.txt will always run before the test task.

```
# Tasks are defined in dodo.py
# To list doit tasks from command line
poetry run doit list
# To run all tasks in parallel (recommended before pushing and/or
# committing)
# 8 is the number of cpu cores, change as wanted
# -v 0 sets verbosity to very low. (Errors will always still be printed.)
poetry run doit -n 8 -v 0
```

- `nox`
  - `nox` is a replacement for `tox`. Both are made to create reproducible Python environments for testing, making docs locally, etc.

```
# To list available nox sessions
# Sessions are defined in noxfile.py
poetry run nox --list
```

- `copier`
  - `copier` is a project templater. Many Python projects follow a similar framework for testing, creating documentations and overall placement of files and configuration. `copier` allows creating a template project

(e.g., https://github.com/nialov/nialov-py-template) which can be firstly cloned as the framework for your own package and secondly to pull updates from the template to your already started project.

```
# To pull copier update from github/nialov/nialov-py-template
poetry run copier update
```

- `pytest`
  - `pytest` is a Python test runner. It is used to run defined tests to check that the package executes as expected. The defined tests in `./tests` contain many regression tests (done with `pytest-regressions`) that make it almost impossible to add features to `fractopo` that changes the results of functions and methods.

```
# To run tests implemented in ./tests directory and as doctests
# within project itself:
poetry run pytest
```

- `coverage`

```
# To check coverage of tests
# (Implemented as nox session!)
poetry run nox --session test_pip
```

- `sphinx`
  - Creates documentation from files in `./docs_src`.

```
# To create documentation
# (Implemented as nox session!)
poetry run nox --session docs
```

Big thanks to all maintainers of the above packages!

6.1 License

6.1.1 Notebook - Fractopo – KB11 Fracture Network Analysis

`fractopo` enables fast and scalable analysis of two-dimensional georeferenced fracture and lineament datasets. These are typically created with remote sensing using a variety of background materials: fractures can be extracted from outcrop orthomosaics and lineaments from digital elevation models (DEMs). Collectively both lineament and fracture datasets can be referred to as *trace* datasets or as *fracture networks*.

`fractopo` implements suggestions for structural geological studies by Peacock and Sanderson (2018):

> **Basic geological descriptions should be followed by measuring their geometries and topologies, understanding their age relationships,** kinematic and mechanics, and developing a realistic, data-led model for related fluid flow.

`fractopo` characterizes the individual and overall geometry and topology of fractures and the fracture network. Furthermore the age relations are investigated with determined topological cross-cutting and abutting relationship between fracture sets.

Whether `fractopo` evolves to implement all the steps in the quote remains to be seen! The functionality grows as more use cases require implementation.

Development imports (just skip to next heading, Data, if not interested!)

Avoid cluttering outputs with warnings.

[1]:
```
import warnings

warnings.filterwarnings("ignore", message="The Shapely GEOS")
warnings.filterwarnings("ignore", message="In a future version, ")
warnings.filterwarnings("ignore", message="No data for colormapping provided via")
```

`geopandas` is the main module which `fractopo` is based on. It along with `shapely` and `pygeos` implement all spatial operations required for two-dimensional fracture network analysis. `geopandas` further implements all input-output operations like reading and writing spatial datasets (shapefiles, GeoPackages, GeoJSON, etc.).

[2]:
```
import geopandas as gpd
```

`geopandas` uses `matplotlib` for visualizing spatial datasets.

[3]:
```
import matplotlib.pyplot as plt
```

During local development the imports might be a bit messy as seen here. Not interesting for end-users.

[4]:
```
# This cell's contents only for development purposes.
from importlib.util import find_spec

if find_spec("fractopo") is None:
    import sys

    sys.path.append("../../")
```

Data

Fracture network data consists of georeferenced lineament or fracture traces, manually or automatically digitized, and a target area boundary that delineates the area in which fracture digiziting has been done. The boundary is important to handle edge effects in network analysis. `fractopo` only has a stub (and untested) implementation for cases where no target area is given so I strongly recommend to always delineate the traced fractures and pass the target area to `Network`.

`geopandas` is used to read and write spatial datasets. Here we use `geopandas` to both download and load trace and area datasets that are hosted on GitHub. A more typical case is that you have local files you wish to analyze in which case you can replace the url string with a path to the local file. E.g.

```
# Local trace data
trace_data_url = "~/data/traces.gpkg"
```

The example dataset here is from an island south of Loviisa, Finland. The island only consists of outcrop quite well polished by glaciations. The dataset is in `ETRS-TM35FIN` coordinate reference system.

[5]:
```
# Trace and target area data available on GitHub
trace_data_url = "https://raw.githubusercontent.com/nialov/fractopo/master/tests/sample_
```

(continues on next page)

(continued from previous page)

```
↪data/KB11/KB11_traces.geojson"
area_data_url = "https://raw.githubusercontent.com/nialov/fractopo/master/tests/sample_
↪data/KB11/KB11_area.geojson"

# Use geopandas to load data from urls
traces = gpd.read_file(trace_data_url)
area = gpd.read_file(area_data_url)

# Name the dataset
name = "KB11"
```

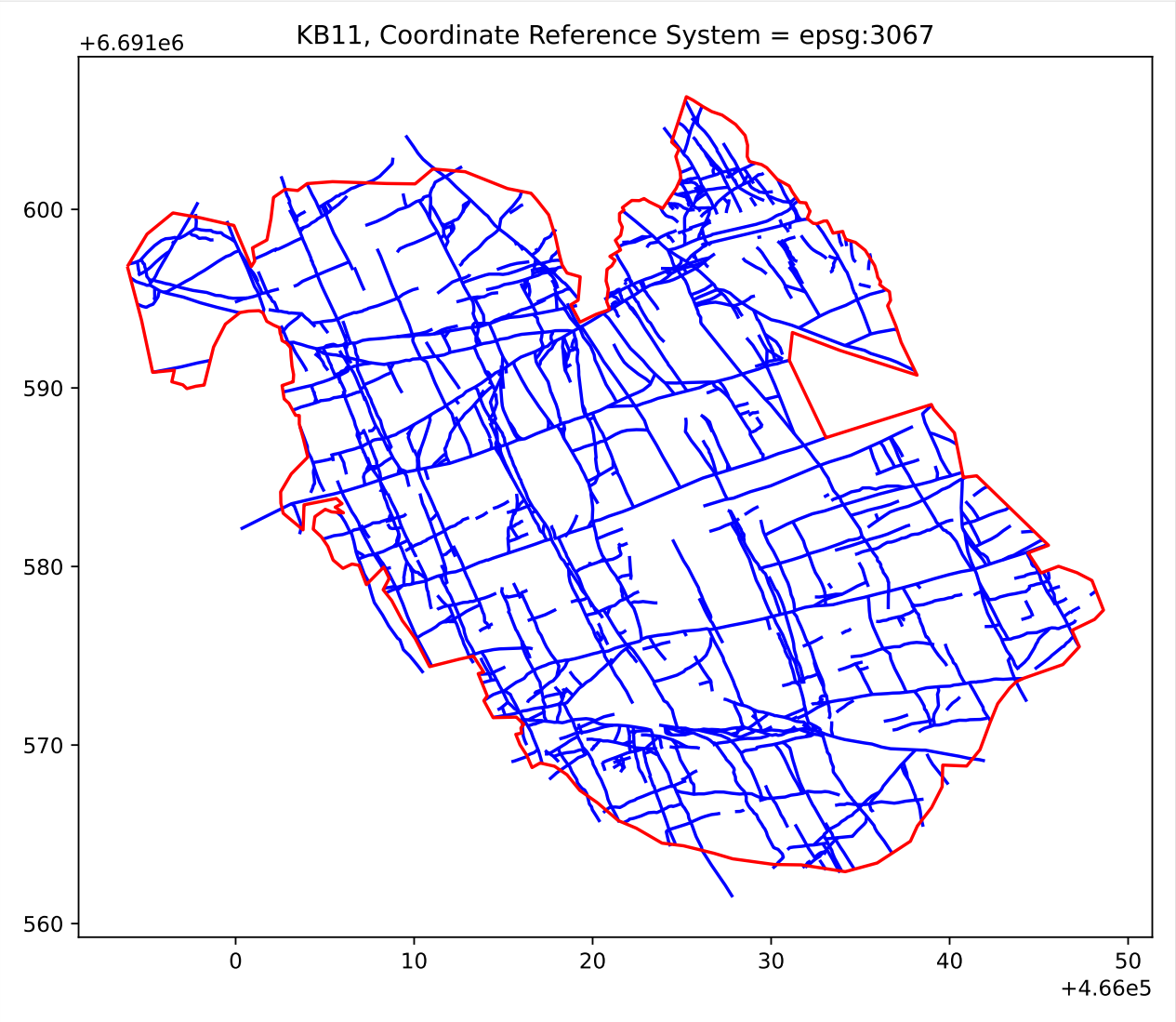
Visualizing trace map data

geopandas has easy methods for plotting spatial data along with data coordinates. The plotting is based on matplotlib.

[6]:
```
# Initialize the figure and ax in which data is plotted
fig, ax = plt.subplots(figsize=(9, 9))

# Plot the loaded trace dataset consisting of fracture traces.
traces.plot(ax=ax, color="blue")

# Plot the loaded area dataset that consists of a single polygon that delineates the␣
↪traces.
area.boundary.plot(ax=ax, color="red")

# Give the figure a title
ax.set_title(f"{name}, Coordinate Reference System = {traces.crs}")
```

```
/home/docs/checkouts/readthedocs.org/user_builds/fractopo/envs/stable/lib/python3.8/site-
↪packages/traitlets/traitlets.py:3437: FutureWarning: --rc={'figure.dpi': 96} for dict-
↪traits is deprecated in traitlets 5.0. You can pass --rc <key=value> ... multiple␣
↪times to add items to a dict.
  warn(
```

[6]:
```
Text(0.5, 1.0, 'KB11, Coordinate Reference System = epsg:3067')
```



Network

So far we have not used any `fractopo` functionality, just `geopandas`. Now we use the `Network` class to create `Network` instances that can be thought of as abstract representations of fracture networks. The fracture network contains traces and a target area boundary delineating the traces.

To characterize the topology of a fracture network `fractopo` determines the topological branches and nodes (Sanderson and Nixon 2015).

- Nodes consist of trace endpoints which can be isolated or snapped to end at another trace.
- Branches consist of every trace segment between the aforementioned nodes.

Automatic determination of branches and nodes is determined with the `determine_branches_nodes` keyword. If given as `False`, they are not determined. You can still use the `Network` object to investigate geometric properties of just the traces.

`Network` initialization should be supplied with information regarding the trace dataset:

- `truncate_traces`

  - If you wish to only characterize the network within the target area boundary, the input traces should be cropped to the target area boundary. This is done when `truncate_traces` is given as `True`. `True` recommended.

- `circular_target_area`
  - If the target area is a circle `circular_target_area` should be given as `True`. A circular target area is recommended to avoid orientation bias in node counting.
- `snap_threshold`
  - To determine topological relationships between traces the abutments between traces should be snapped to some tolerance. This tolerance can be given here, in the same unit used for the traces. It represents the smallest distance between nodes for which `fractopo` interprets them as two distinct nodes. This value should be carefully chosen depending on the size of the area of interest. As a reference, when digitizing in QGIS with snapping turned on, the tolerance is probably much lower than `0.001`.
  - The trace validation functionality of `fractopo` can be (and should be) used to check that there are no topological errors within a certain tolerance.

[7]:
```
# Import the Network class from fractopo
from fractopo import Network
```

[8]:
```
# Create Network and automatically determine branches and nodes
# The Network instance is saved as kb11_network variable.
kb11_network = Network(
    traces,
    area,
    name=name,
    determine_branches_nodes=True,
    truncate_traces=True,
    circular_target_area=True,
    snap_threshold=0.001,
)
```

Visualizing fracture network branches and nodes

We can similarly to the traces visualize the branches and nodes with `geopandas` plotting.

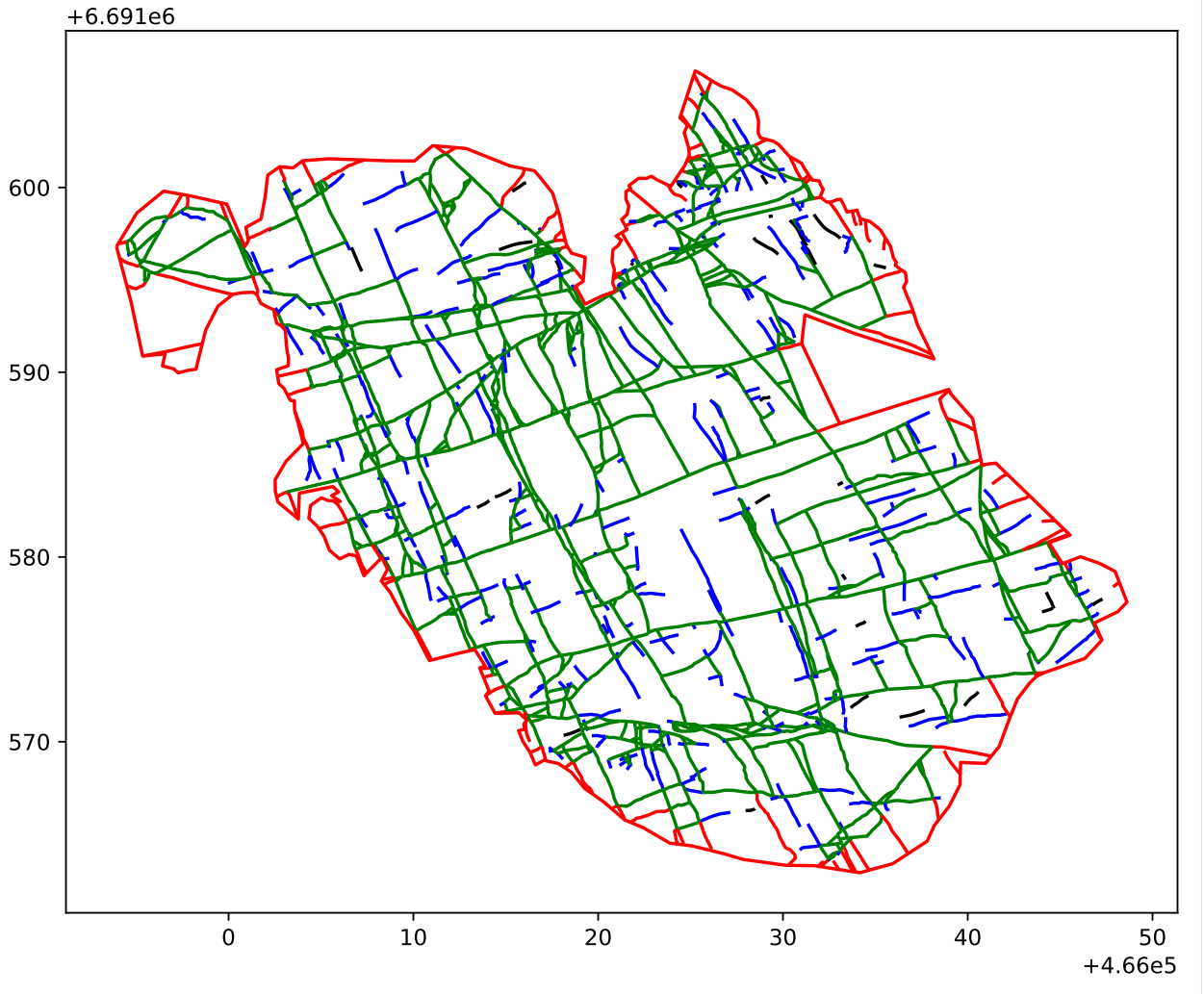
[9]:
```
# Import identifier strings of topological branches and nodes
from fractopo.general import CC_branch, CI_branch, II_branch, X_node, Y_node, I_node


# Function to determine color for each branch and node type
def assign_colors(feature_type: str):
    if feature_type in (CC_branch, X_node):
        return "green"
    if feature_type in (CI_branch, Y_node):
        return "blue"
    if feature_type in (II_branch, I_node):
        return "black"
    return "red"
```

Branch or Node Type	Color
C - C, X	Green
C - I, Y	Blue
I - I, I	Black
Other	Red

Branches

[10]:
```
fix, ax = plt.subplots(figsize=(9, 9))
kb11_network.branch_gdf.plot(
    colors=[assign_colors(bt) for bt in kb11_network.branch_types], ax=ax
)
area.boundary.plot(ax=ax, color="red")
```

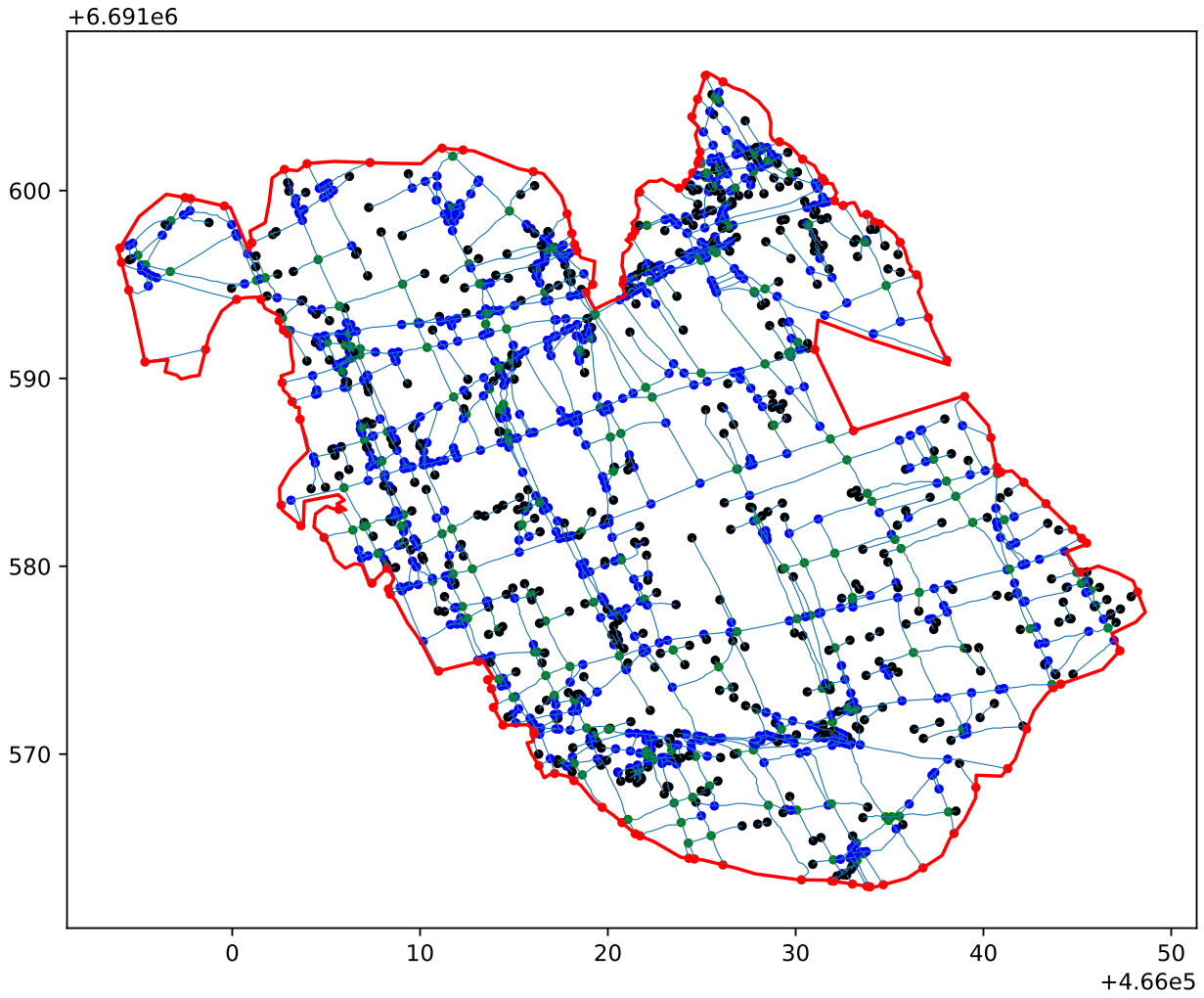
[10]:
```
<Axes: >
```



Nodes

[11]:
```
fix, ax = plt.subplots(figsize=(9, 9))
# Traces
kb11_network.trace_gdf.plot(ax=ax, linewidth=0.5)
# Nodes
kb11_network.node_gdf.plot(
    c=[assign_colors(bt) for bt in kb11_network.node_types], ax=ax, markersize=10
)
area.boundary.plot(ax=ax, color="red")
```

[11]:
```
<Axes: >
```



Geometric Fracture Network Characterization

The most basic geometric properties of traces are their **length** and **orientation**.

Length is the overall travel distance along the digitized trace. The length of traces individually is usually not interesting but the value **distribution** of all of the lengths is (Bonnet et al. 2001). `fractopo` uses another Python package, `powerlaw`, for determining power-law, lognormal and exponential distribution fits. The wrapper around `powerlaw` is thin and therefore I urge you to see its documentation and associated article for more info.

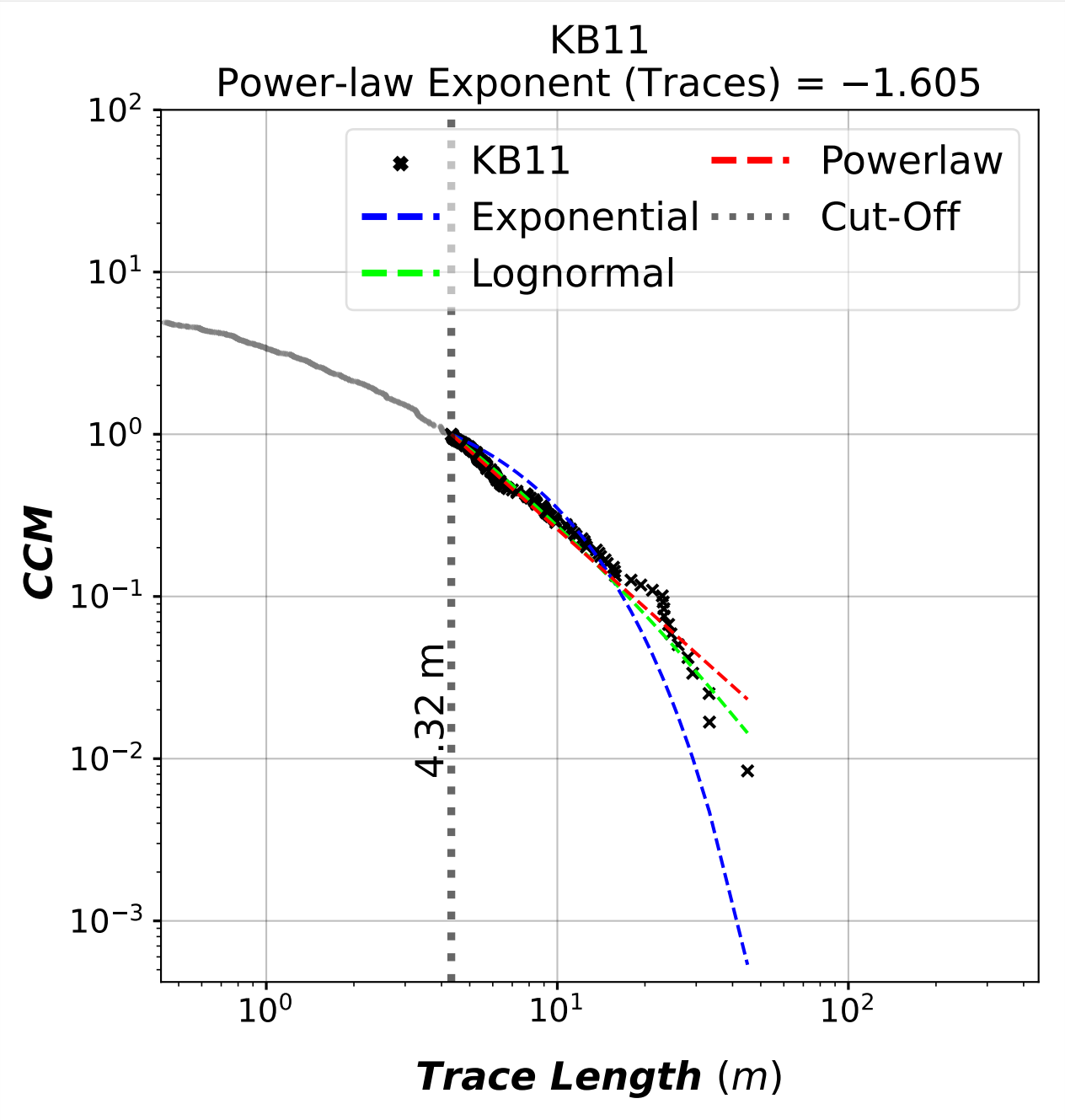
Orientation of a trace (or branch, or any line) can be defined in multiple ways that approach the same result when the line is **sublinear**:

- Draw a straight line between the start and endpoints of the trace and calculate the orientation of that line.
  - This is the approach used in `fractopo`. Simple, but when the trace is curvy enough the simplification might be detrimental to analysis.
- Plot each coordinate point of a trace and fit a linear regression trend line. Calculate the orientation of the trend line.
- Calculate the orientation of each segment between coordinate points resulting in multiple orientation values for a single trace.

Length distributions

Traces

[12]:
```
# Plot length distribution fits (powerlaw, exponential and lognormal) of fracture traces␣
↪using powerlaw
fit, fig, ax = kb11_network.plot_trace_lengths()
```

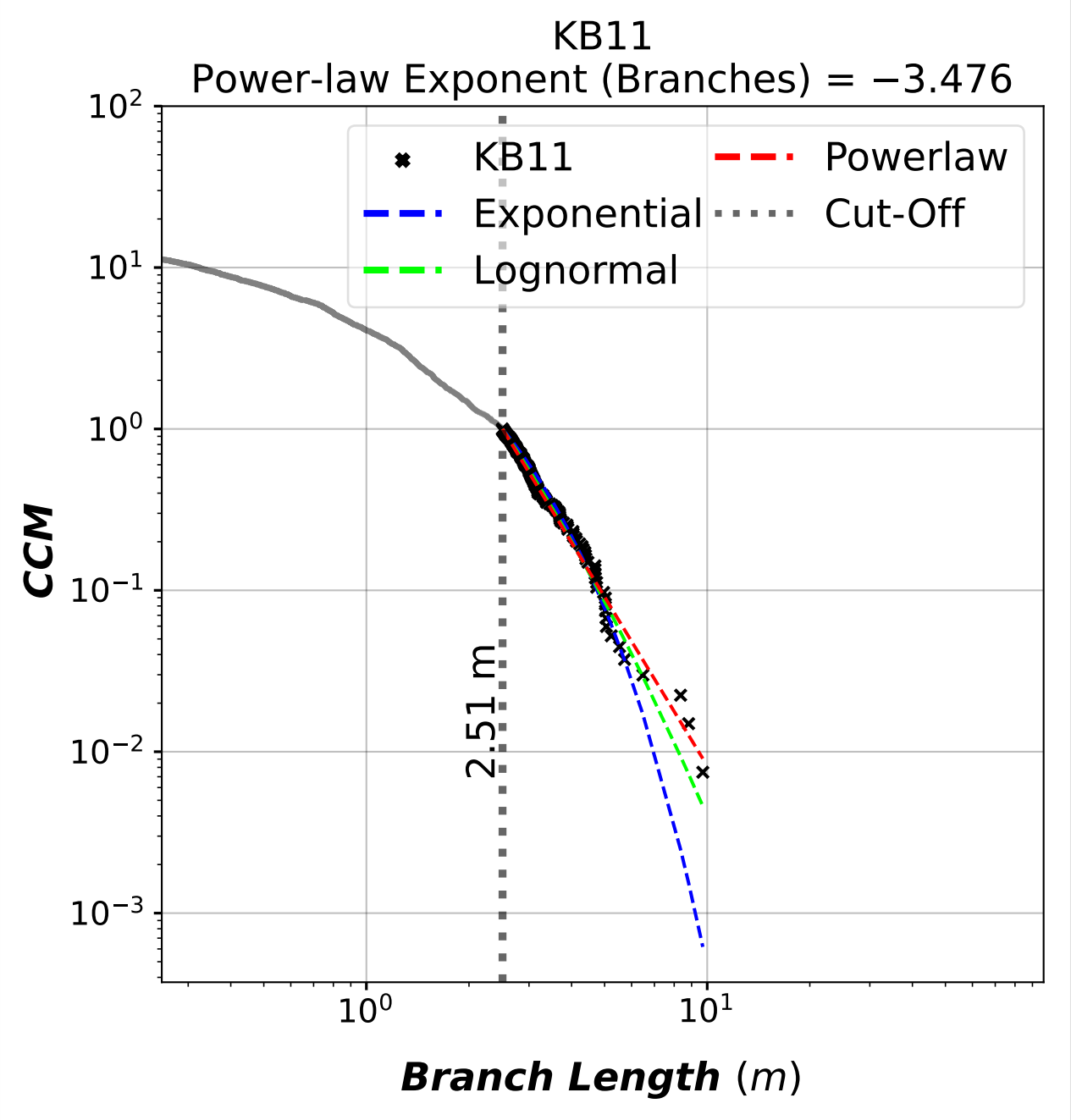


[13]:
```
# Fit properties
print(f"Automatically determined powerlaw cut-off: {fit.xmin}")
print(f"Powerlaw exponent: {fit.alpha - 1}")
print(
    f"Compare powerlaw fit to lognormal: R, p = {fit.distribution_compare('power_law',
↪'lognormal')}"
)
```

```
Automatically determined powerlaw cut-off: 4.321478841775524
Powerlaw exponent: 1.6051888040287916
Compare powerlaw fit to lognormal: R, p = (-0.6749942925041177, 0.41051593411959586)
```

Branches

[14]:
```
# Length distribution of branches
fit, fig, ax = kb11_network.plot_branch_lengths()
```



[15]:
```
# Fit properties
print(f"Automatically determined powerlaw cut-off: {fit.xmin}")
print(f"Powerlaw exponent: {fit.alpha - 1}")
print(
    f"Compare powerlaw fit to lognormal: R, p = {fit.distribution_compare('power_law',
↪'lognormal')}"
```

(continues on next page)

(continued from previous page)

```
)
```

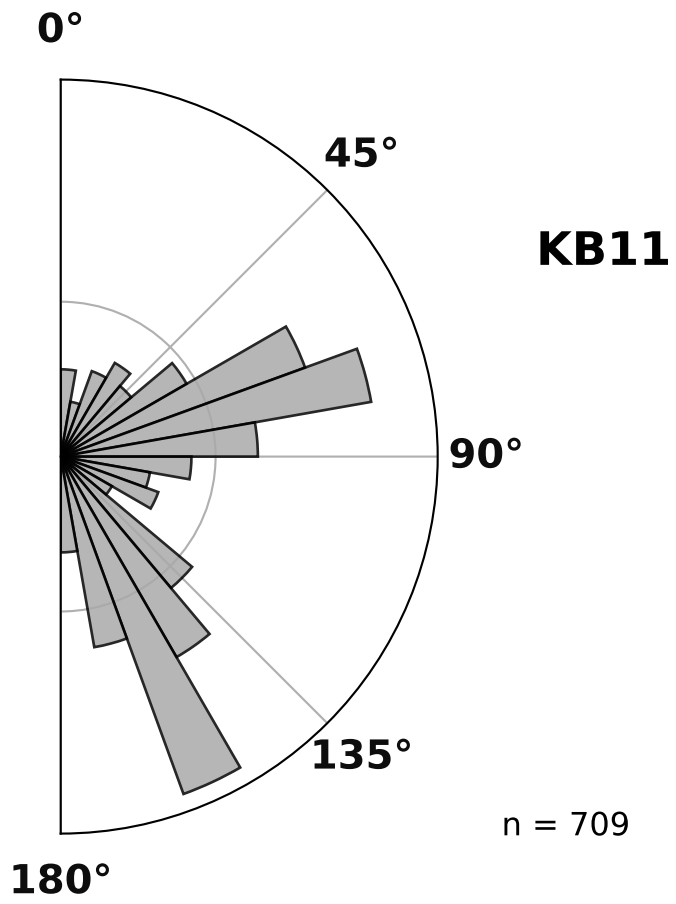
```
Automatically determined powerlaw cut-off: 2.5085762870774153
Powerlaw exponent: 3.475884087746053
Compare powerlaw fit to lognormal: R, p = (-0.48507540234571334, 0.5591241023355218)
```

Rose plots

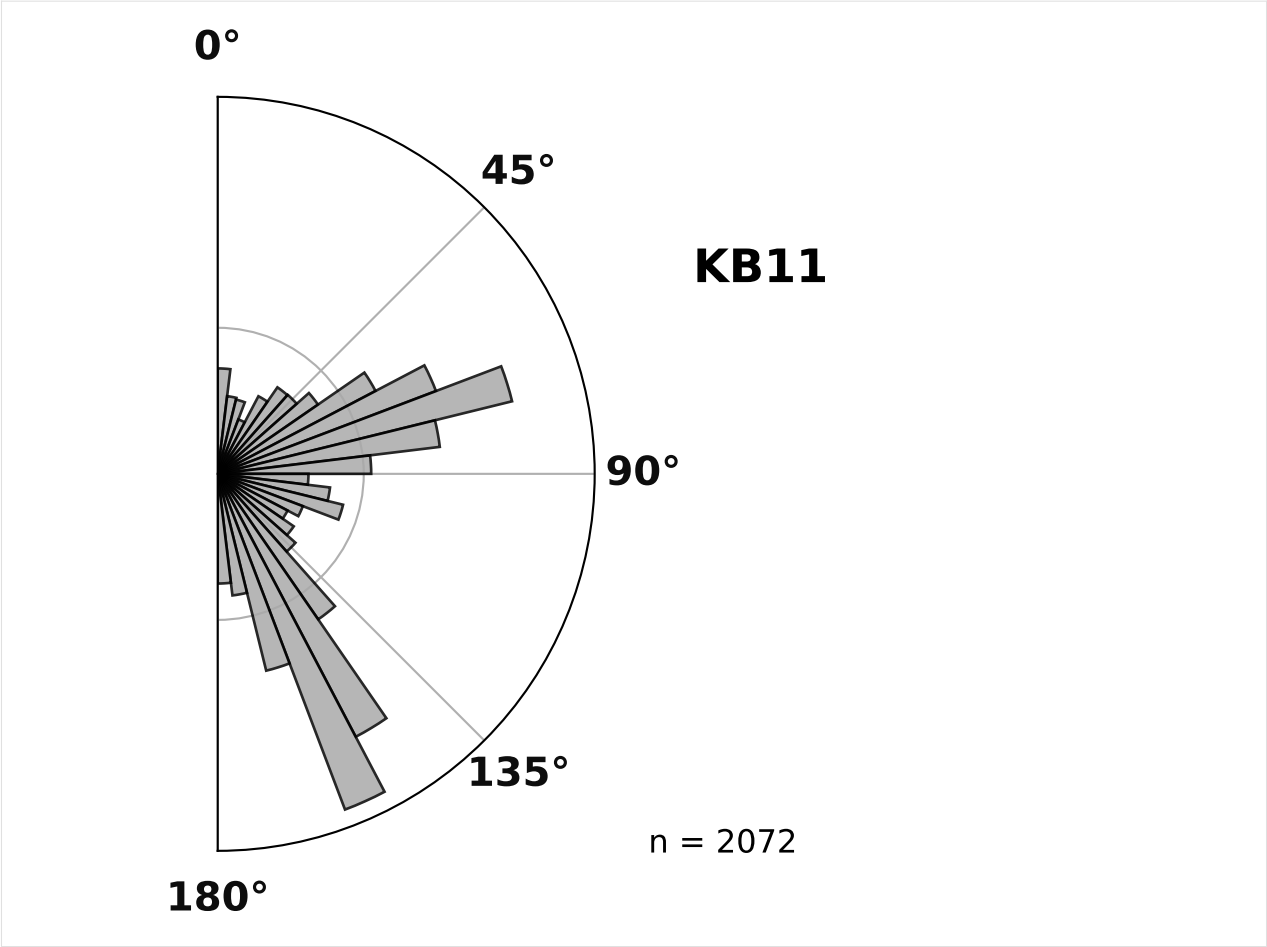
A rose plot is a histogram where the bars have been oriented based on pre-determined bins. `fractopo` rose plots are length-weighted and equal-area. Length-weighted means that each bin contains the total length of traces or branches within the orientation range of the bin.

The method for calculating the bins and reasoning for using **equal-area** rose plots is from publication by Sanderson and Peacock (2020).

[16]:
```
# Plot azimuth rose plot of fracture traces and branches
azimuth_bin_dict, fig, ax = kb11_network.plot_trace_azimuth()
```



[17]:
```
# Plot azimuth rose plot of fracture branches
azimuth_bin_dict, fig, ax = kb11_network.plot_branch_azimuth()
```



Topological Fracture Network Characterization

The determination of branches and nodes are essential for characterizing the topology of a fracture network. The topology is the collection of properties of the traces that do not change when the traces are transformed continously i.e. the traces are not cut but are extended or shrinked. In geological terms the traces can go through ductile transformation without losing their topological properties but not brittle transformation. Furthermore this means the end topology of the traces is a result of brittle transformation(s).

At its simplest the proportion of different types of branches and nodes are used to characterize the topology.

Branches can be categorized into three main categories:

- **C–C** is connected at both endpoints
- **C-I** is connected at one endpoint
- **I-I** is not connected at either endpoint

Nodes can be similarly categorized into three categories:

- **X** represents intersection between two traces
- **Y** represents abutment of one trace to another
- **I** represents isolated termination of a trace

Furthermore **E** node and any **E**-containing branch classification (e.g. **I-E**) are related to the trace area boundary. Branches are always cropped to the boundary and branches that are cut then have a **E** node as end endpoint.

Node and branch proportions

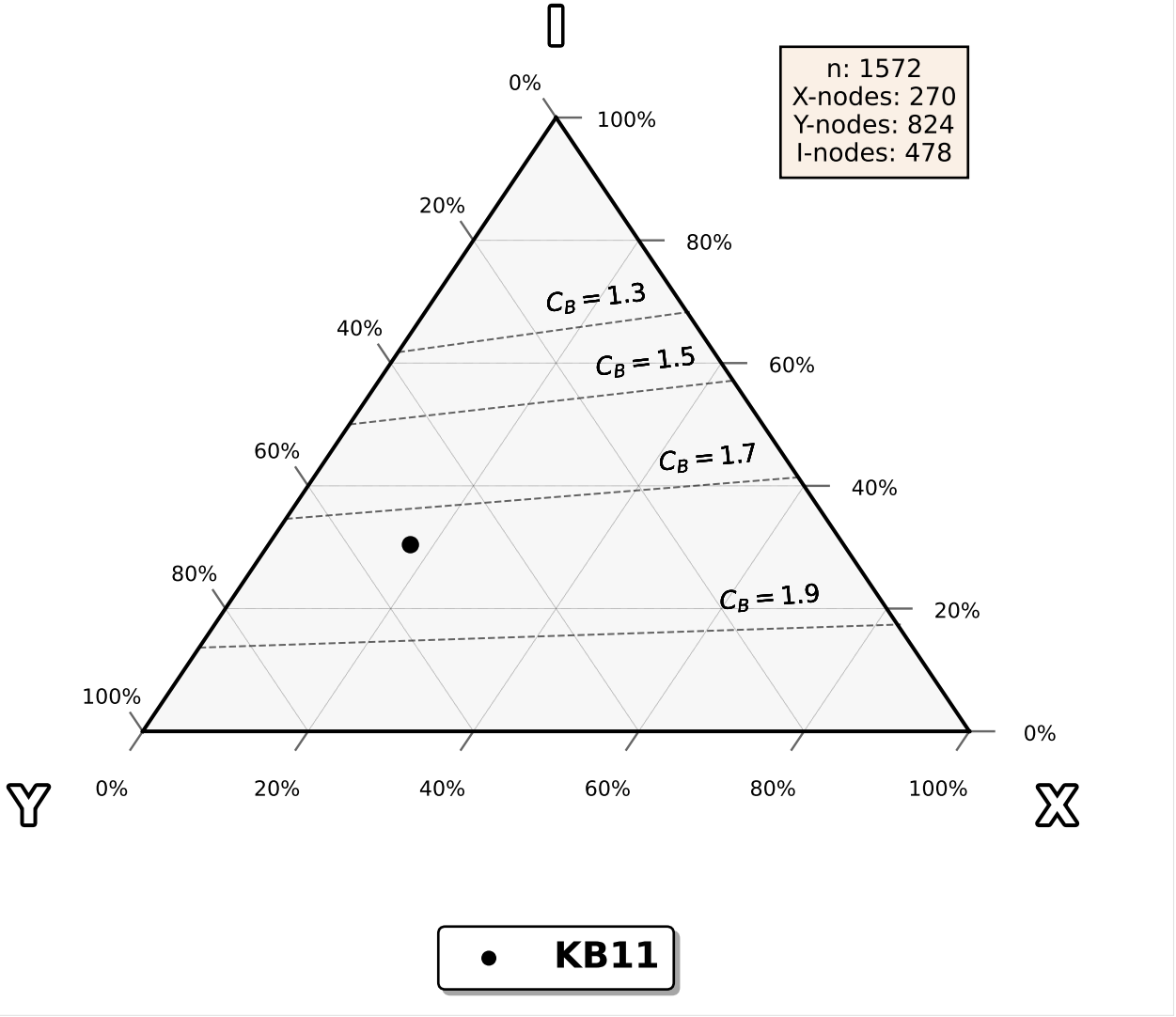
The proportion of the different types of nodes and branches have direct implications for the overall connectivity of a fracture network (Sanderson and Nixon 2015).

The proportions are plotted on ternary plots. The plotting uses python-ternary.

[18]:
```
kb11_network.node_counts
```

[18]:
```
{'X': 270, 'Y': 824, 'I': 478, 'E': 114}
```

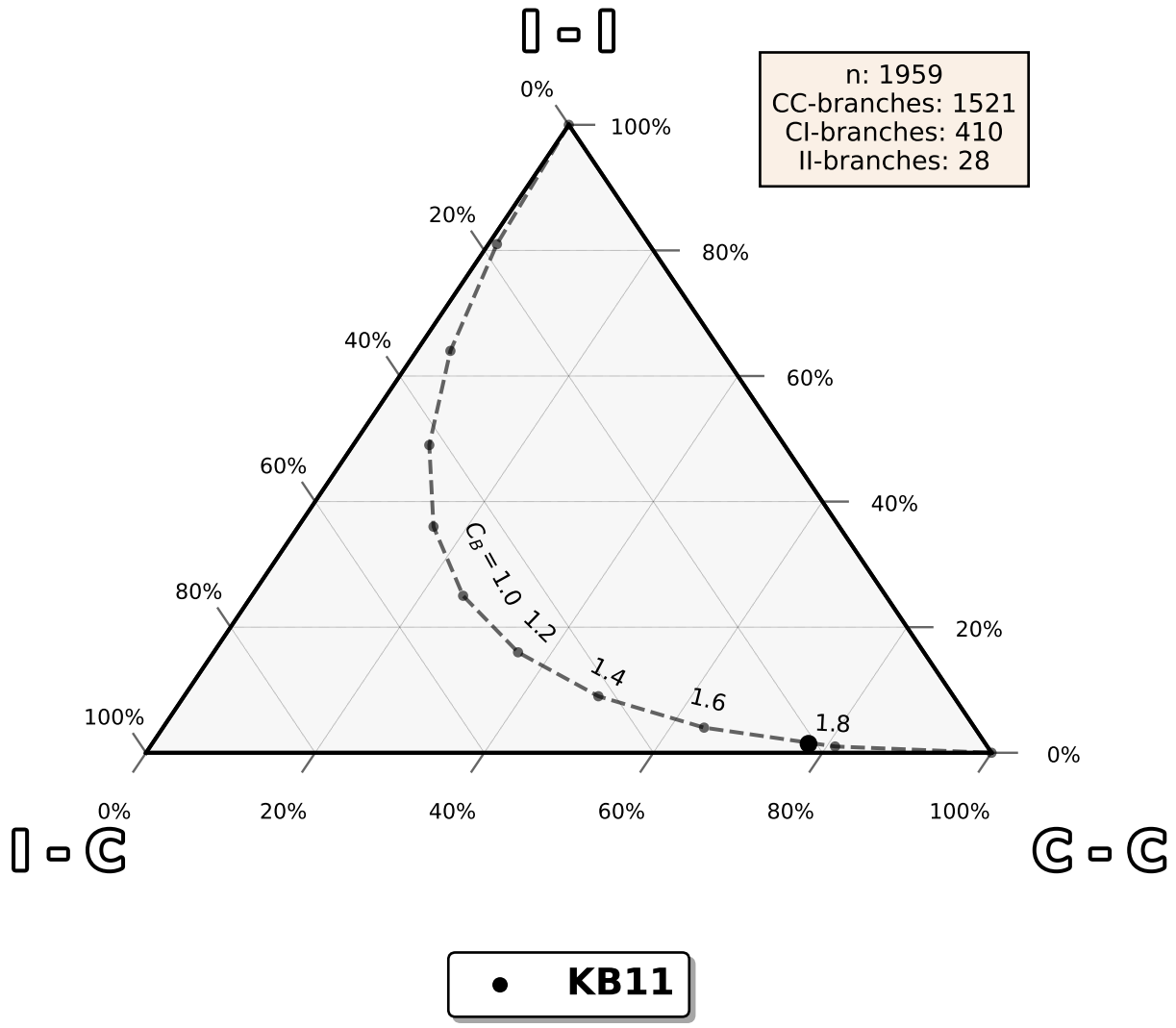
[19]:
```
# Plot ternary XYI-node proportion plot
fig, ax, tax = kb11_network.plot_xyi()
```



[20]:
```
kb11_network.branch_counts
```

[20]:
```
{'C - C': 1521,
 'C - I': 410,
 'I - I': 28,
 'C - E': 100,
 'I - E': 12,
 'E - E': 1}
```

[21]:
```
# Plot ternary branch (C-C, C-I, I-I) proportion plot
fig, ax, tax = kb11_network.plot_branch()
```



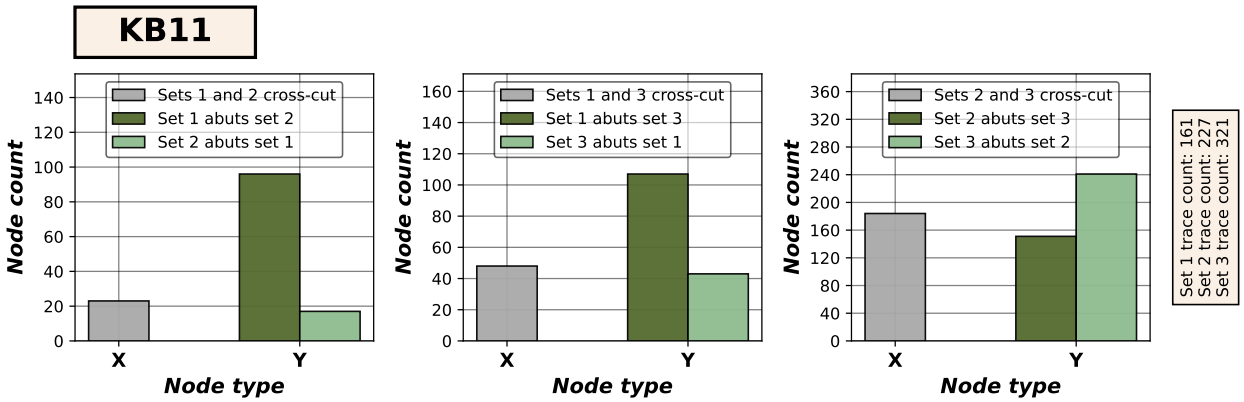
Crosscutting and abutting relationships

If the geometry and topology of the fracture network are investigated together the cross-cutting and abutting relationships between orientation-binned fracture sets can be determined. Traces can be binned into sets based on their orientation (e.g. N-S oriented traces could belong to Set 1 and E-W oriented traces to Set 2). If the endpoints of the traces of sets are examined the abutment relationship between can be determined i.e. which abuts which (e.g. does the N-S oriented Set 1 traces abut to E-W oriented Set 2 or do they crosscut each other equal amounts.)

[22]:
```
# Sets are defaults
print(f"Azimuth set names: {kb11_network.azimuth_set_names}")
print(f"Azimuth set ranges: {kb11_network.azimuth_set_ranges}")
```

```
Azimuth set names: ('1', '2', '3')
Azimuth set ranges: ((0, 60), (60, 120), (120, 180))
```

[23]:
```
# Plot crosscutting and abutting relationships between azimuth sets
figs, fig_axes = kb11_network.plot_azimuth_crosscut_abutting_relationships()
```



Numerical Fracture Network Characterization Parameters

The quantity, total length and other geometric and topological properties of the traces, branches and nodes within the target area can be determined as numerical values. For the following parameters I refer you to the following articles:

- Mauldon et al. 2001
- Sanderson and Nixon 2015

[24]:
```
kb11_network.parameters
```

[24]:
```
{'Fracture Intensity B21': 1.2633728710295158,
 'Fracture Intensity P21': 1.2633728710295158,
 'Trace Min Length': 5.820766091346741e-10,
 'Trace Max Length': 33.1085306416674,
 'Trace Mean Length': 2.2058247378420526,
 'Dimensionless Intensity P22': 2.786779132035443,
 'Area': 1237.9003657531266,
 'Number of Traces (Real)': 709,
 'Number of Traces': 651.0,
 'Branch Min Length': 0.006639501944314653,
 'Branch Max Length': 5.721558550556387,
 'Branch Mean Length': 0.7761437911315212,
```

(continues on next page)

(continued from previous page)

```
 'Areal Frequency B20': 1.627756203766927,
 'Areal Frequency P20': 0.5258904658323919,
 'Dimensionless Intensity B22': 0.9805590097335628,
 'Connections per Trace': 3.3609831029185866,
 'Connections per Branch': 1.7627791563275435,
 'Fracture Intensity (Mauldon)': 1.4357438039436288,
 'Fracture Density (Mauldon)': 0.5258904658323919,
 'Trace Mean Length (Mauldon)': 2.7301194777720483,
 'Connection Frequency': 0.8837544848243267,
 'Number of Branches': 2015.0,
 'Number of Branches (Real)': 2072}
```

Contour Grids

To visualize the spatial variation of geometric and topological parameter values the network can be sampled with a grid of rectangles. From the center of each rectangle a sampling circle is made which is used to do the actual sampling following Nyberg et al. 2018 (See Fig. 3F).

Sampling with a contour grid is time-consuming and therefore the following code is not executed within this notebooks by default. The end result is embedded as images. Paste the code from the below cell blocks to a Python cell to execute them.

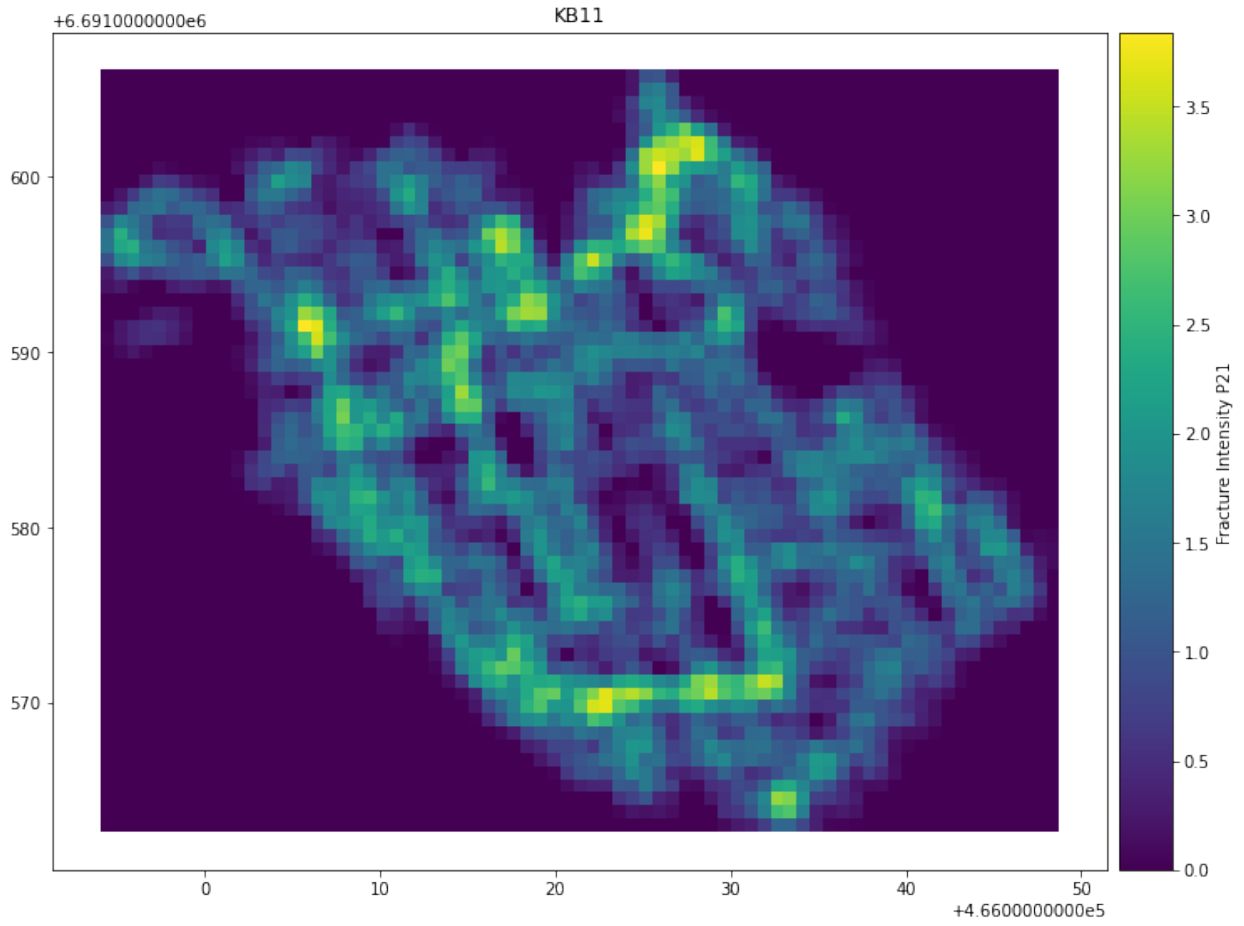
To perform sampling with a cell width of 0.75 m:

```
sampled_grid = kb11_network.contour_grid(
    cell_width=0.75,
)
```

To visualize results for parameter *Fracture Intensity P21*:

```
kb11_network.plot_contour(parameter="Fracture Intensity P21", sampled_grid=sampled_grid)
```

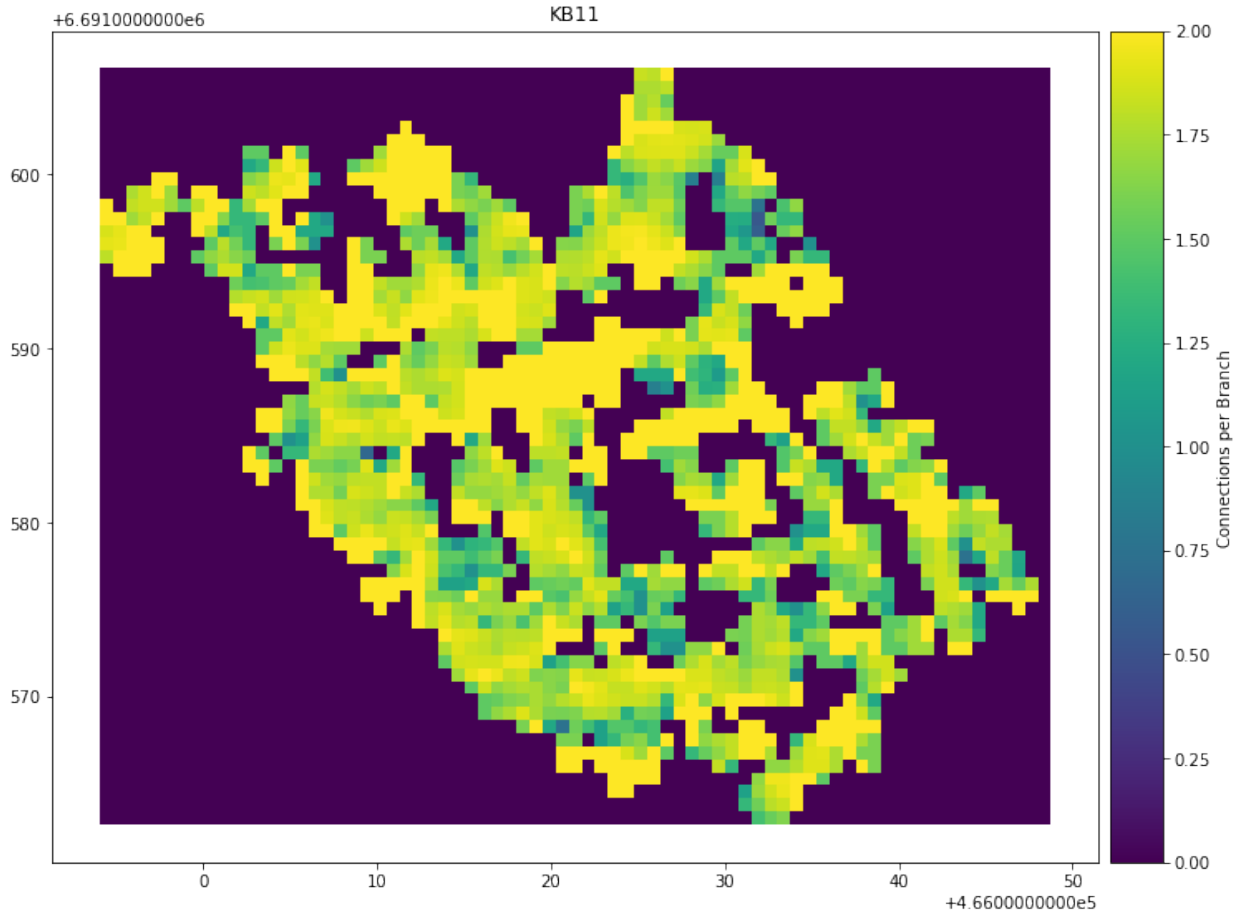
Result:



To visualize results for parameter *Connections per Branch*:

```
kb11_network.plot_contour(parameter="Connections per Branch", sampled_grid=sampled_grid)
```

Result:



Tests to verify notebook consistency

[25]:
```
assert hasattr(kb11_network, "contour_grid") and hasattr(kb11_network, "plot_contour")
```

6.1.2 Notebook - Fractopo – KB11 Trace Data Validation

[1]:
```
import warnings

warnings.filterwarnings("ignore", message="The Shapely GEOS")
warnings.filterwarnings("ignore", message="In a future version, ")
warnings.filterwarnings("ignore", message="No data for colormapping provided via")
```

[2]:
```
import geopandas as gpd
```

[3]:
```
# This cell's contents only for development purposes.
from importlib.util import find_spec

if find_spec("fractopo") is None:
    import sys
```

(continues on next page)

(continued from previous page)

```
    sys.path.append("../../")
```

[4]:
```
from fractopo import Validation
import matplotlib.pyplot as plt

plt.close()
```

Data (KB11)

[5]:
```
# Trace and target area data available on GitHub
trace_data_url = "https://raw.githubusercontent.com/nialov/fractopo/master/tests/sample_
↪data/KB11/KB11_traces.geojson"
area_data_url = "https://raw.githubusercontent.com/nialov/fractopo/master/tests/sample_
↪data/KB11/KB11_area.geojson"

# Use geopandas to load data from urls
traces = gpd.read_file(trace_data_url)
area = gpd.read_file(area_data_url)

# Name the dataset
name = "KB11"
```

Validation (KB11)

[6]:
```
# Create validation object with fixing (i.e. modification of data) allowed.
kb11_validation = Validation(traces, area, name=name, allow_fix=True)
```

[7]:
```
# Run actual validation and capture the outputted validated trace GeoDataFrame
kb11_validated = kb11_validation.run_validation()
```

Validation results (KB11)

[8]:
```
# Normal DataFrame methods are available for data inspection
kb11_validated.columns
```

[8]:
```
Index(['Name', 'Shape_Leng', 'geometry', 'VALIDATION_ERRORS'], dtype='object')
```

[9]:
```
# Convert column data to string to allow hashing and return all unique
# validation errors.
kb11_validated["VALIDATION_ERRORS"].astype(str).unique()
```

[9]:
```
array(['()', "('SHARP TURNS',)"], dtype=object)
```

[10]:
```
# Better description function is found in fractopo.cli
from fractopo.cli import describe_results

describe_results(kb11_validated, kb11_validation.ERROR_COLUMN)
```

```
Out of 707 traces, 1 were invalid.
```

```
There were 1 error types. These were:
SHARP TURNS
```

The KB11 dataset only contains `SHARP TURNS` errors which are normally non-disruptive in further analyses.

See documentation: https://fractopo.readthedocs.io/en/latest/validation/errors.html

6.1.3 Notebook - Fractopo – KB7 Trace Data Validation

[1]:
```
import warnings

warnings.filterwarnings("ignore", message="The Shapely GEOS")
warnings.filterwarnings("ignore", message="In a future version, ")
warnings.filterwarnings("ignore", message="No data for colormapping provided via")
```

[2]:
```
import geopandas as gpd
```

[3]:
```
# This cell's contents only for development purposes.
from importlib.util import find_spec

if find_spec("fractopo") is None:
    import sys

    sys.path.append("../../")
```

[4]:
```
from fractopo import Validation
import matplotlib.pyplot as plt

plt.close()
```

Data (KB7)

[5]:
```
# Trace and target area data available on GitHub
trace_data_url = "https://raw.githubusercontent.com/nialov/fractopo/master/tests/sample_
↪data/KB7/KB7_traces.geojson"
area_data_url = "https://raw.githubusercontent.com/nialov/fractopo/master/tests/sample_
↪data/KB7/KB7_area.geojson"

# Use geopandas to load data from urls
traces = gpd.read_file(trace_data_url)
area = gpd.read_file(area_data_url)

# Name the dataset
name = "KB7"
```

Validation (KB7)

[6]:
```
# Create validation object with fixing (i.e. modification of data) allowed.
# AREA_EDGE_SNAP_MULTIPLIER is overridden to keep catching this error even with future␣
↪default
# value changes
kb7_validation = Validation(
    traces, area, name=name, allow_fix=True, AREA_EDGE_SNAP_MULTIPLIER=2.5
)
```

[7]:
```
# Run actual validation and capture the outputted validated trace GeoDataFrame
kb7_validated = kb7_validation.run_validation()
```

Validation results (KB7)

[8]:
```
# Normal DataFrame methods are available for data inspection
kb7_validated.columns
```

[8]:
```
Index(['Name', 'Shape_Leng', 'geometry', 'VALIDATION_ERRORS'], dtype='object')
```

[9]:
```
# Convert column data to string to allow hashing and return all unique
# validation errors.
kb7_validated["VALIDATION_ERRORS"].astype(str).unique()
```

[9]:
```
array(['()', "('MULTI JUNCTION',)", "('MULTI JUNCTION', 'V NODE')",
       "('SHARP TURNS',)", "('TRACE UNDERLAPS TARGET AREA',)"],
      dtype=object)
```

[10]:
```
# Better description function is found in fractopo.cli
from fractopo.cli import describe_results

describe_results(kb7_validated, kb7_validation.ERROR_COLUMN)
```

```
Out of 240 traces, 7 were invalid.
```

```
There were 4 error types. These were:
MULTI JUNCTION
SHARP TURNS
V NODE
TRACE UNDERLAPS TARGET AREA
```

The KB7 dataset contains the above errors of which `MULTI JUNCTION` and `TRACE UNDERLAPS TARGET AREA` are disruptive in further analysis.

See documentation: https://fractopo.readthedocs.io/en/latest/validation/errors.html

Visualization of errors in notebook

Though visualization here is possible, GIS-software (e.g. QGIS, ArcGIS) are much more interactive and are recommended for actual fixing and further error inspection.

MULTI JUNCTION

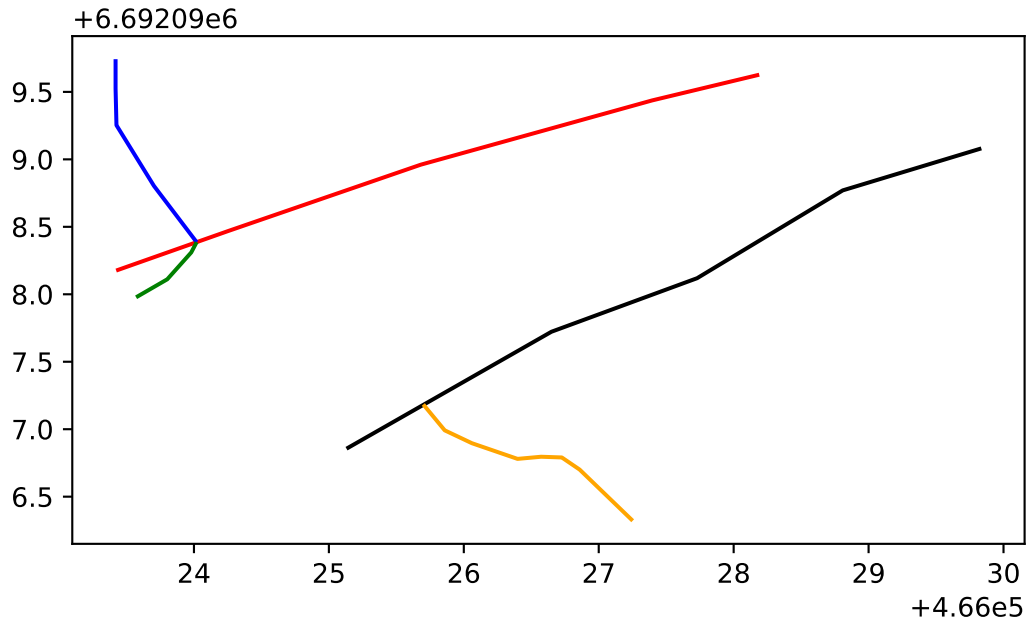
[11]:
```
# Find MULTI JUNCTION erroneous traces in GeoDataFrame
kb7_multijunctions = kb7_validated.loc[
    ["MULTI JUNCTION" in err for err in kb7_validated[kb7_validation.ERROR_COLUMN]]
]
kb7_multijunctions
```

[11]:
```
     Name  Shape_Leng                                           geometry  \
168  None    4.985900  LINESTRING (466023.424 6692098.176, 466024.252...
169  None    5.232445  LINESTRING (466025.131 6692096.856, 466026.650...
171  None    1.541532  LINESTRING (466024.020 6692098.386, 466023.703...
206  None    1.864415  LINESTRING (466025.700 6692097.183, 466025.858...
219  None    0.619474  LINESTRING (466023.571 6692097.978, 466023.802...

            VALIDATION_ERRORS
168         (MULTI JUNCTION,)
169         (MULTI JUNCTION,)
171  (MULTI JUNCTION, V NODE)
206         (MULTI JUNCTION,)
219  (MULTI JUNCTION, V NODE)
```

[12]:
```
kb7_multijunctions.plot(colors=["red", "black", "blue", "orange", "green"])
```

```
/home/docs/checkouts/readthedocs.org/user_builds/fractopo/envs/stable/lib/python3.8/site-
↪packages/traitlets/traitlets.py:3437: FutureWarning: --rc={'figure.dpi': 96} for dict-
↪traits is deprecated in traitlets 5.0. You can pass --rc <key=value> ... multiple␣
↪times to add items to a dict.
  warn(
```

[12]:
```
<Axes: >
```

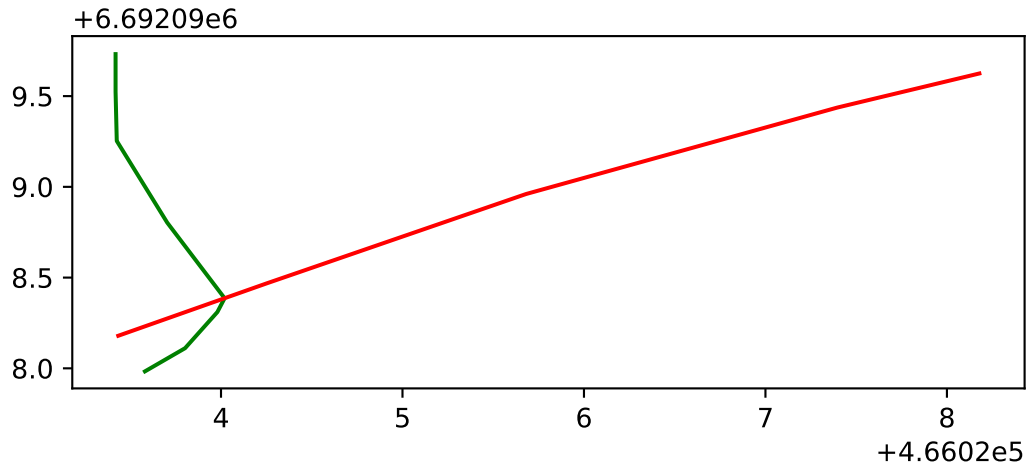


The plot shows that the green and blue traces abut at their endpoints which is not a valid topology for traces. The fix is done by merging the green and blue traces.

Additionally the orange trace has a dangling end instead of being accurately snapped to the black trace.

[13]:
```
# Example fix for blue and green traces
from shapely.ops import linemerge

gpd.GeoSeries(
    [
        linemerge(
            [kb7_multijunctions.geometry.iloc[4], kb7_multijunctions.geometry.iloc[2]]
        ),
        kb7_multijunctions.geometry.iloc[0],
    ]
).plot(colors=["green", "red"])
```

[13]:
```
<Axes: >
```



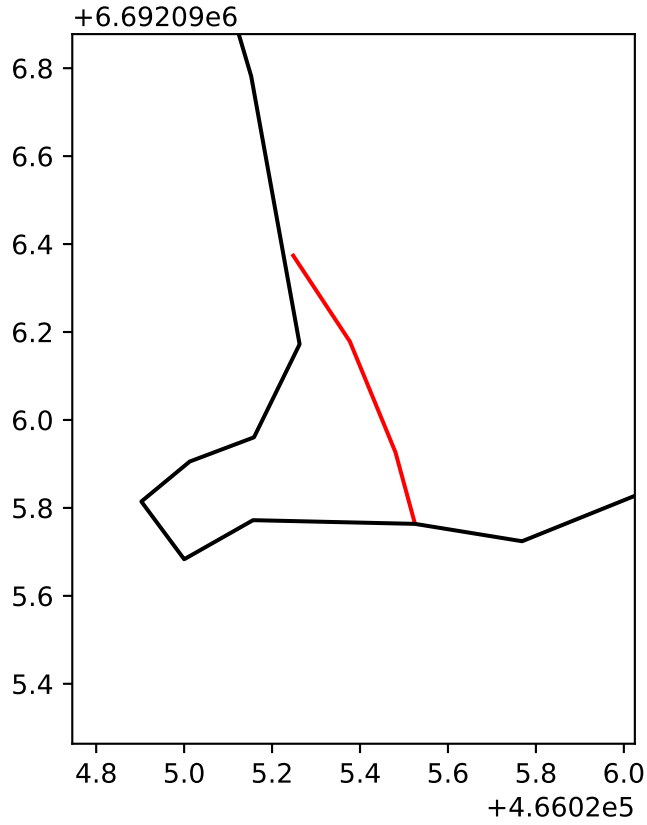
TRACE UNDERLAPS TARGET AREA

[14]:
```
# Find TRACE UNDERLAPS TARGET AREA erroneous traces in GeoDataFrame
kb7_underlaps = kb7_validated.loc[
    [
        "TRACE UNDERLAPS TARGET AREA" in err
        for err in kb7_validated[kb7_validation.ERROR_COLUMN]
    ]
]
kb7_underlaps
```

[14]:
```
     Name  Shape_Leng                                           geometry  \
207  None    0.679687  LINESTRING (466025.246 6692096.378, 466025.377...

                  VALIDATION_ERRORS
207  (TRACE UNDERLAPS TARGET AREA,)
```

[15]:
```
# Create figure, ax base
fig, ax = plt.subplots()

# Plot the underlapping trace along with the trace area boundary
kb7_underlaps.plot(ax=ax, color="red")
area.boundary.plot(ax=ax, color="black")

# Get trace bounds
minx, miny, maxx, maxy = kb7_underlaps.total_bounds

ax.set_xlim(minx - 0.5, maxx + 0.5)
ax.set_ylim(miny - 0.5, maxy + 0.5)
```

[15]:
```
(6692095.263678445, 6692096.877513003)
```



The plot shows that the trace underlaps the target area at least on the northern end and maybe on the southern end. The fix is implemented by extending the trace to meet the target area boundary.

[ ]:

6.1.4 Gallery of fractopo example scripts and/or plots

All matplotlib plots can be saved with:

```
fig.savefig("savename.png", bbox_inches="tight")
# Or
plt.savefig("savename.png", bbox_inches="tight")
```

Where `savename` can be replaced with name/path of where you want to save the figure. `bbox_inches` is given to make sure the whole plot is saved even thought individual elements go outside the `matplotlib` figure bounding box. `png` extension can be replaced with e.g. `svg`. See https://matplotlib.org/ for more information about plotting.

Plotting topological ternary plots with `fractopo`

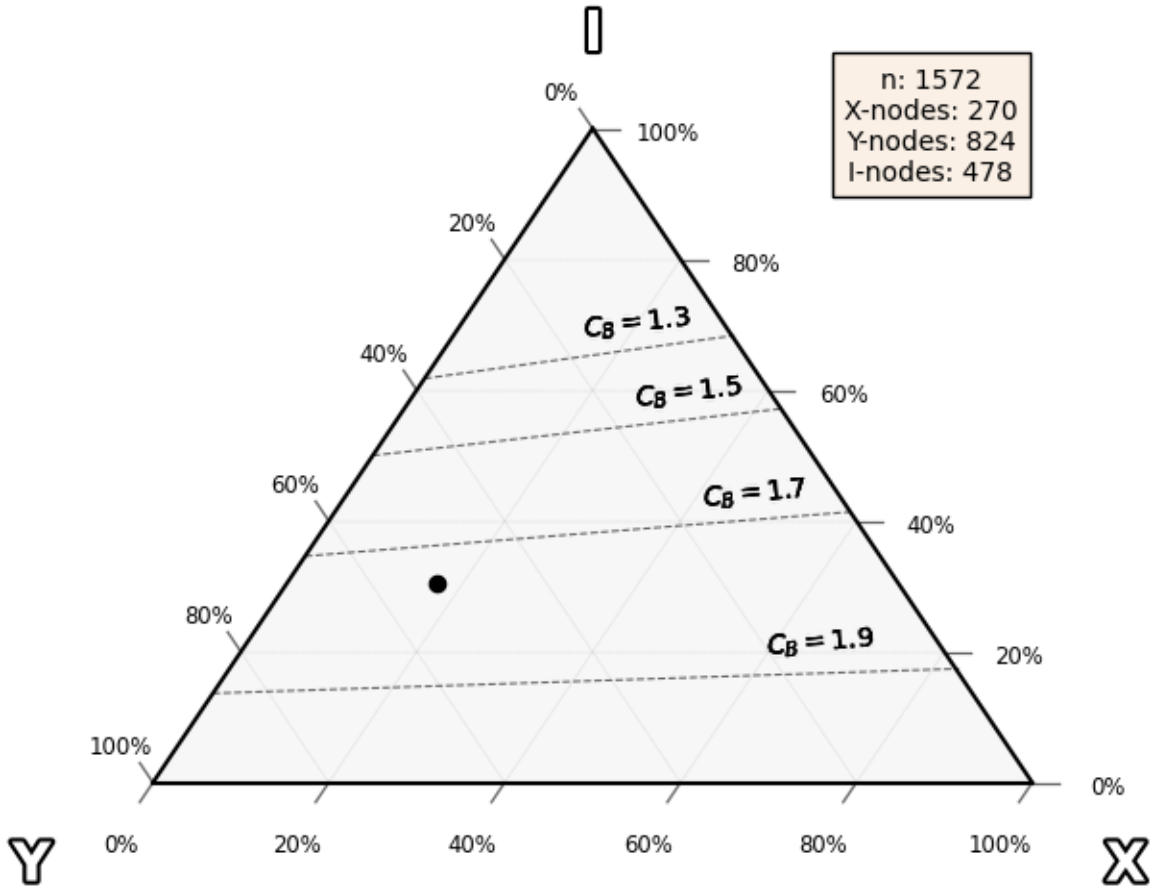
Initializing

```
from pprint import pprint

# Load kb11_network network from examples/example_networks.py
from example_networks import kb11_network
```

```
/home/docs/checkouts/readthedocs.org/user_builds/fractopo/envs/stable/lib/python3.8/site-
↪packages/geopandas/tools/clip.py:67: FutureWarning: In a future version, `df.iloc[:,␣
↪i] = newvals` will attempt to set the values inplace instead of always setting a new␣
↪array. To retain the old behavior, use either `df[df.columns[i]] = newvals` or, if␣
↪columns are non-unique, `df.isetitem(i, newvals)`
  clipped.loc[
/home/docs/checkouts/readthedocs.org/user_builds/fractopo/envs/stable/lib/python3.8/site-
↪packages/geopandas/tools/clip.py:67: FutureWarning: In a future version, `df.iloc[:,␣
↪i] = newvals` will attempt to set the values inplace instead of always setting a new␣
↪array. To retain the old behavior, use either `df[df.columns[i]] = newvals` or, if␣
↪columns are non-unique, `df.isetitem(i, newvals)`
  clipped.loc[
/home/docs/checkouts/readthedocs.org/user_builds/fractopo/envs/stable/lib/python3.8/site-
↪packages/geopandas/tools/clip.py:67: FutureWarning: In a future version, `df.iloc[:,␣
↪i] = newvals` will attempt to set the values inplace instead of always setting a new␣
↪array. To retain the old behavior, use either `df[df.columns[i]] = newvals` or, if␣
↪columns are non-unique, `df.isetitem(i, newvals)`
  clipped.loc[
```

Plotting a ternary plot of fracture network node counts

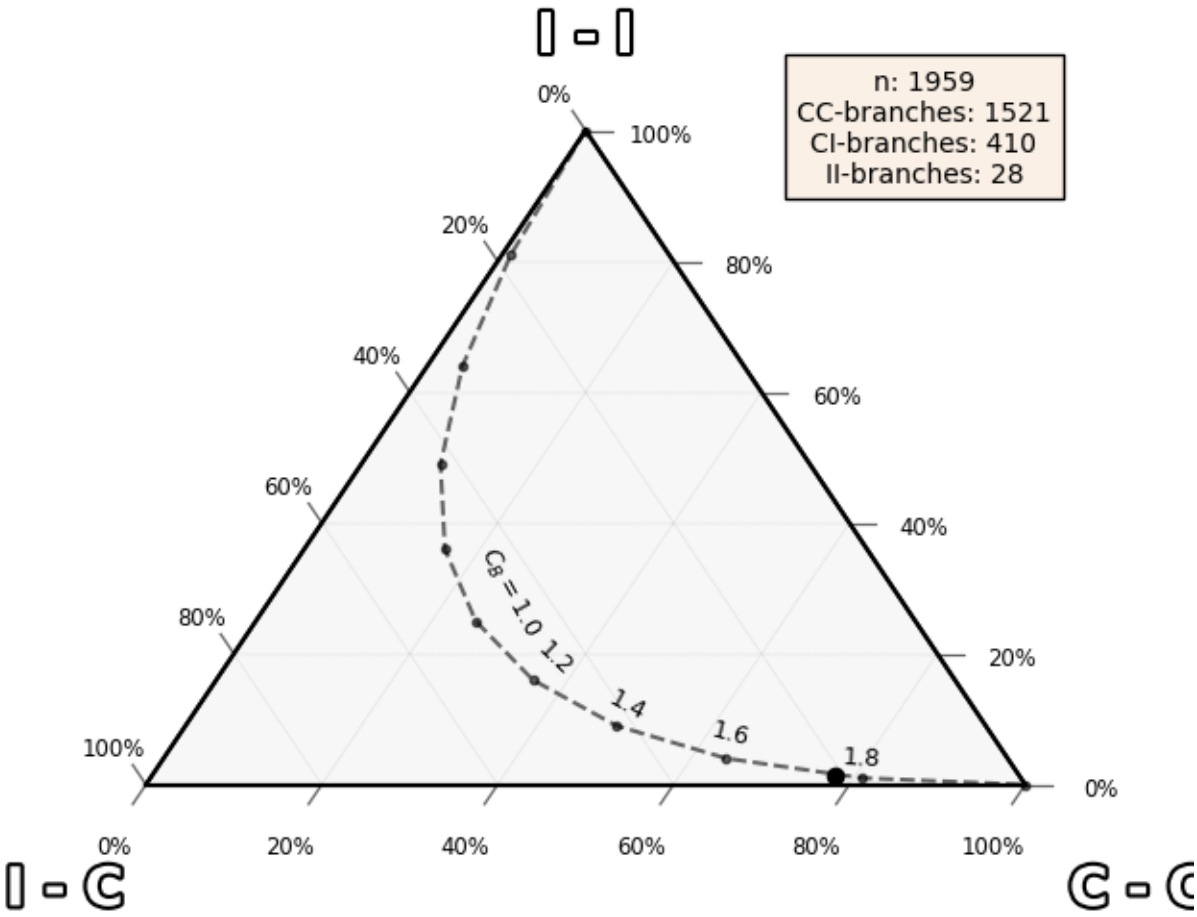
```
xyi_fig, xyi_ax, xyi_tax = kb11_network.plot_xyi()
```




```
/home/docs/checkouts/readthedocs.org/user_builds/fractopo/envs/stable/lib/python3.8/site-
↪packages/ternary/plotting.py:148: UserWarning: No data for colormapping provided via 'c
↪'. Parameters 'vmin', 'vmax' will be ignored
  ax.scatter(xs, ys, vmin=vmin, vmax=vmax, **kwargs)
```

Plotting a ternary plot of fracture network branch counts

```
branch_fig, branch_ax, branch_tax = kb11_network.plot_branch()
```




```
/home/docs/checkouts/readthedocs.org/user_builds/fractopo/envs/stable/lib/python3.8/site-
↪packages/ternary/plotting.py:148: UserWarning: No data for colormapping provided via 'c
↪'. Parameters 'vmin', 'vmax' will be ignored
  ax.scatter(xs, ys, vmin=vmin, vmax=vmax, **kwargs)
```

Numerical data is accessible

```
pprint(dict(node_counts=kb11_network.node_counts))

pprint(dict(branch_counts=kb11_network.branch_counts))
```

```
{'node_counts': {'E': 114, 'I': 478, 'X': 270, 'Y': 824}}
{'branch_counts': {'C - C': 1521,
                   'C - E': 100,
                   'C - I': 410,
                   'E - E': 1,
                   'I - E': 12,
                   'I - I': 28}}
```

Total running time of the script: (0 minutes 20.834 seconds)

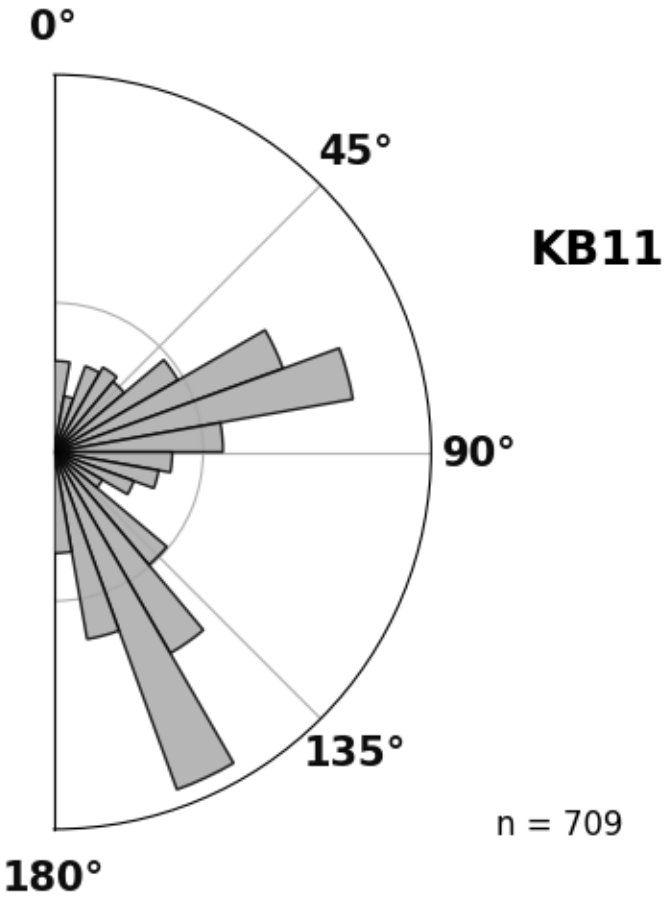
Plotting rose plots with `fractopo`

Initializing

```
from pprint import pprint

# Load kb11_network network from examples/example_networks.py
from example_networks import kb11_network
```

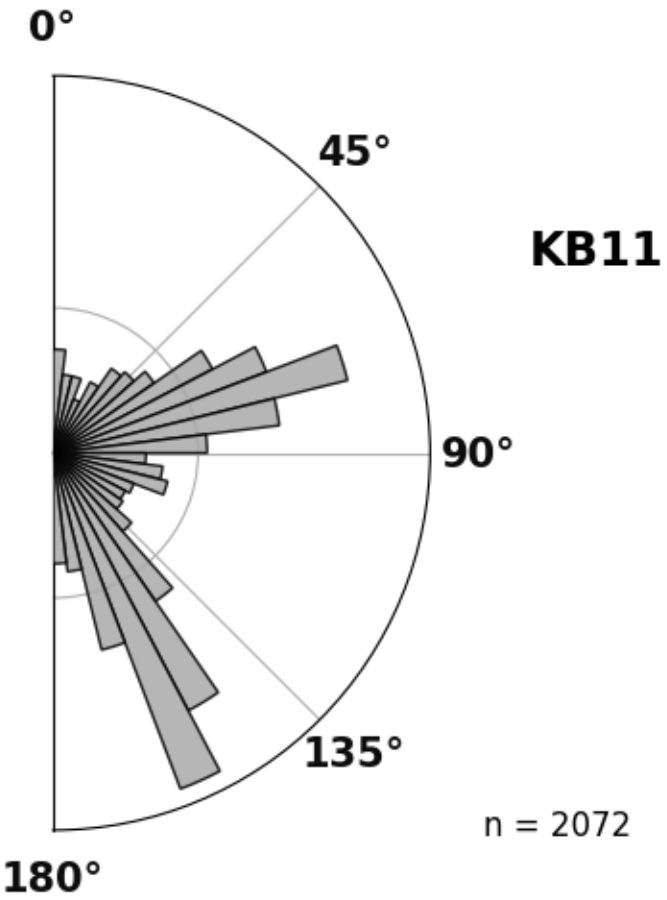
Plotting a rose plot of fracture network trace orientations

```
# Rose plot of network trace orientations
azimuth_bins, fig, ax = kb11_network.plot_trace_azimuth()
```



Plotting a rose plot of fracture network branch orientations

```
# Rose plot of network branch orientations
kb11_network.plot_branch_azimuth()
```




```
(AzimuthBins(bin_width=6.923076923076923, bin_locs=array([  3.46153846,  10.38461538, ␣
↪17.30769231,  24.23076923,
        31.15384615,  38.07692308,  45.        ,  51.92307692,
        58.84615385,  65.76923077,  72.69230769,  79.61538462,
        86.53846154,  93.46153846, 100.38461538, 107.30769231,
       114.23076923, 121.15384615, 128.07692308, 135.        ,
       141.92307692, 148.84615385, 155.76923077, 162.69230769,
       169.61538462, 176.53846154]), bin_heights=array([ 23.85186242,  13.85935962,  13.
↪88406748,   7.36702792,
        14.23952299,  23.40760406,  25.99082394,  33.56150329,
        72.13951553, 115.79880919, 204.63184844, 115.57635472,
        52.8100409 ,  19.18577615,  27.12296283,  30.95261975,
        16.55924935,  14.77946813,  15.99990073,  24.51727887,
        72.23261948, 188.16867511, 286.88599501,  91.41047876,
        31.76997234,  27.22640228])), <Figure size 650x510 with 1 Axes>, <PolarAxes:␣
↪title={'center': 'KB11'}>)
```

Numerical data is accessible with methods and class properties

```
pprint((kb11_network.branch_azimuth_set_counts, kb11_network.trace_azimuth_set_counts))
```

```
({'E-W': 1045, 'N-S': 1027}, {'E-W': 315, 'N-S': 394})
```

The azimuth sets were not explicitly given during Network creation so they are set to defaults.

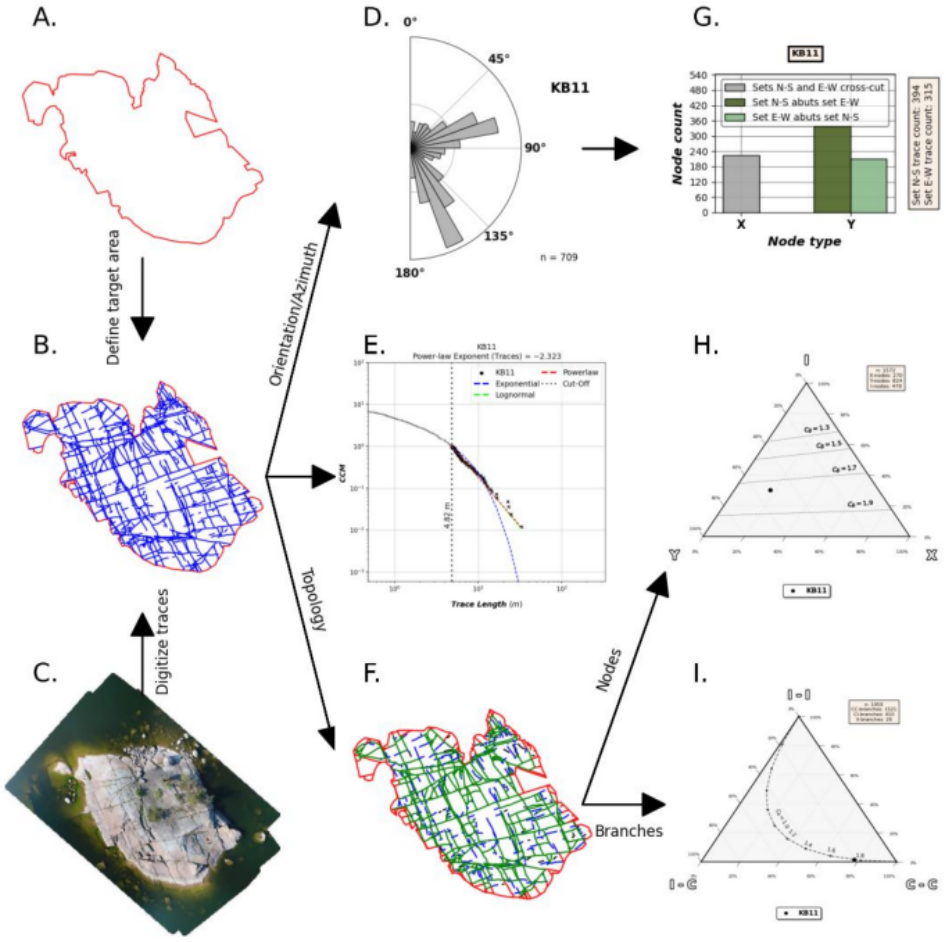
```
pprint((kb11_network.azimuth_set_names, kb11_network.azimuth_set_ranges))
```

```
(('N-S', 'E-W'), ((135, 45), (45, 135)))
```

Total running time of the script: (0 minutes 0.464 seconds)

Workflow visualisation of `fractopo`

See `examples/fractopo_workflow_visualisation.py` for the code.




```
({'E-W': 1045, 'N-S': 1027}, {'E-W': 315, 'N-S': 394})
(('N-S', 'E-W'), ((135, 45), (45, 135)))
{'trace exponential Kolmogorov-Smirnov distance D': 0.16734893835952314,
 'trace exponential lambda': 0.2898919999912063,
 'trace exponential loglikelihood': -188.02077841990496,
 'trace lengths cut off proportion': 0.8815232722143864,
 'trace lognormal Kolmogorov-Smirnov distance D': 0.055285689798228566,
 'trace lognormal loglikelihood': -181.36559959113393,
 'trace lognormal mu': -24.654367132184213,
 'trace lognormal sigma': 3.413632605089442,
 'trace lognormal vs. exponential R': 1.840247791139716,
 'trace lognormal vs. exponential p': 0.06573186649520334,
 'trace power_law Kolmogorov-Smirnov distance D': 0.05261149958762154,
```

(continues on next page)

(continued from previous page)

```
 'trace power_law alpha': 3.323399316187791,
 'trace power_law cut-off': 4.815579557192599,
 'trace power_law exponent': -2.323399316187791,
 'trace power_law sigma': 0.25350364847712353,
 'trace power_law vs. exponential R': 1.792390864166387,
 'trace power_law vs. exponential p': 0.07307037719921719,
 'trace power_law vs. lognormal R': -0.09973967423028064,
 'trace power_law vs. lognormal p': 0.9205510020886868,
 'trace power_law vs. truncated_power_law R': -0.3949163478172287,
 'trace power_law vs. truncated_power_law p': 0.6549247892396883,
 'trace truncated_power_law Kolmogorov-Smirnov distance D': 0.06088951719124519,
 'trace truncated_power_law alpha': 3.101655682133221,
 'trace truncated_power_law exponent': -2.101655682133221,
 'trace truncated_power_law lambda': 0.01693723427355317,
 'trace truncated_power_law loglikelihood': -181.27535493663555}
{'branch exponential Kolmogorov-Smirnov distance D': 0.055610337970091184,
 'branch exponential lambda': 1.2799623045206825,
 'branch exponential loglikelihood': -79.83654595014787,
 'branch lengths cut off proportion': 0.9488416988416989,
 'branch lognormal Kolmogorov-Smirnov distance D': 0.056242712153426244,
 'branch lognormal loglikelihood': -80.11030644664197,
 'branch lognormal mu': 0.3954002513747114,
 'branch lognormal sigma': 0.48573668994587976,
 'branch lognormal vs. exponential R': -1.2860976878834365,
 'branch lognormal vs. exponential p': 0.1984089709536515,
 'branch power_law Kolmogorov-Smirnov distance D': 0.05150595395106594,
 'branch power_law alpha': 4.9790958462418935,
 'branch power_law cut-off': 2.4602426467400713,
 'branch power_law exponent': -3.9790958462418935,
 'branch power_law sigma': 0.38648395404192654,
 'branch power_law vs. exponential R': -1.2796975009064637,
 'branch power_law vs. exponential p': 0.20065154388298556,
 'branch power_law vs. lognormal R': -1.2484090815914064,
 'branch power_law vs. lognormal p': 0.21188128491687686,
 'branch power_law vs. truncated_power_law R': -1.5645011856154882,
 'branch power_law vs. truncated_power_law p': 0.055258061048291784,
 'branch truncated_power_law Kolmogorov-Smirnov distance D': 0.05467641228173603,
 'branch truncated_power_law alpha': 1.0000206602209936,
 'branch truncated_power_law exponent': -2.066022099356246e-05,
 'branch truncated_power_law lambda': 1.0171370451351636,
 'branch truncated_power_law loglikelihood': -79.83784521401718}
('N-S', 'E-W')
((135, 45), (45, 135))
(('N-S', 'E-W'), ((135, 45), (45, 135)))
   name        sets    x    y y-reverse error-count
0  KB11  (N-S, E-W)  224  339       210           0
/home/docs/checkouts/readthedocs.org/user_builds/fractopo/envs/stable/lib/python3.8/site-
↪packages/ternary/plotting.py:148: UserWarning: No data for colormapping provided via 'c
↪'. Parameters 'vmin', 'vmax' will be ignored
  ax.scatter(xs, ys, vmin=vmin, vmax=vmax, **kwargs)
{'node_counts': {'E': 114, 'I': 478, 'X': 270, 'Y': 824}}
{'branch_counts': {'C - C': 1521,
```

(continues on next page)

(continued from previous page)

```
                    'C - E': 100,
                    'C - I': 410,
                    'E - E': 1,
                    'I - E': 12,
                    'I - I': 28}}
Saving workflow plot to /tmp/tmp_8l96557/fractopo_workflow_visualisation.jpg
```

```
from pathlib import Path
from tempfile import TemporaryDirectory

import fractopo_workflow_visualisation
import matplotlib.pyplot as plt
from PIL import Image

with TemporaryDirectory() as tmp_dir:
    fig_output_path = Path(tmp_dir) / "fractopo_workflow_visualisation.jpg"
    fractopo_workflow_visualisation.main(output_path=fig_output_path)

    figure, ax = plt.subplots(1, 1, figsize=(9, 9))
    with Image.open(fig_output_path) as image:
        ax.imshow(image)
        ax.axis("off")
```

Total running time of the script: (0 minutes 7.674 seconds)

Plotting azimuth set relationships

The relationships i.e. crosscuts and abutments between lineament & fracture traces can be determined with `fractopo`.

```
from pprint import pprint

import matplotlib as mpl
import matplotlib.pyplot as plt

# Load kb11_network network from examples/example_networks.py
from example_networks import kb11_network

mpl.rcParams["figure.figsize"] = (5, 5)
mpl.rcParams["font.size"] = 8
```

Analyzing azimuth set relationships

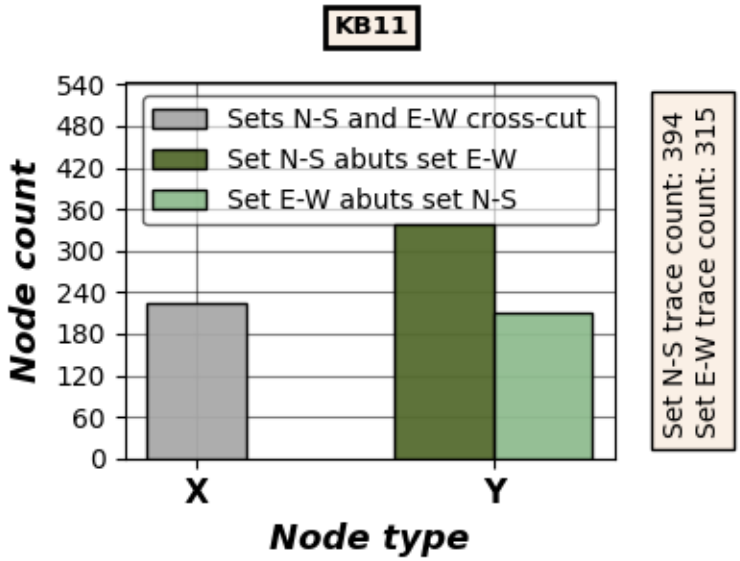
Azimuth sets (set by user):

```
pprint((kb11_network.azimuth_set_names, kb11_network.azimuth_set_ranges))
```

```
(('N-S', 'E-W'), ((135, 45), (45, 135)))
```

Visualize the relationships with a plot.

```
figs, _ = kb11_network.plot_azimuth_crosscut_abutting_relationships()

# Edit the figure to better fit the gallery webpage
figs[0].suptitle(
    kb11_network.name,
    fontsize="large",
    fontweight="bold",
    fontfamily="DejaVu Sans",
)
plt.tight_layout()
plt.show()
```



The relationships are also accessible in numerical form as a `pandas` DataFrame.

```
pprint(kb11_network.azimuth_set_relationships)
```

```
   name        sets    x    y y-reverse error-count
0  KB11  (N-S, E-W)  224  339       210           0
```

Total running time of the script: (0 minutes 0.152 seconds)

Visualizing azimuth sets

Initializing

```
from pprint import pprint

import matplotlib.pyplot as plt

# Load kb11_network network from examples/example_networks.py
from example_networks import kb11_network
```

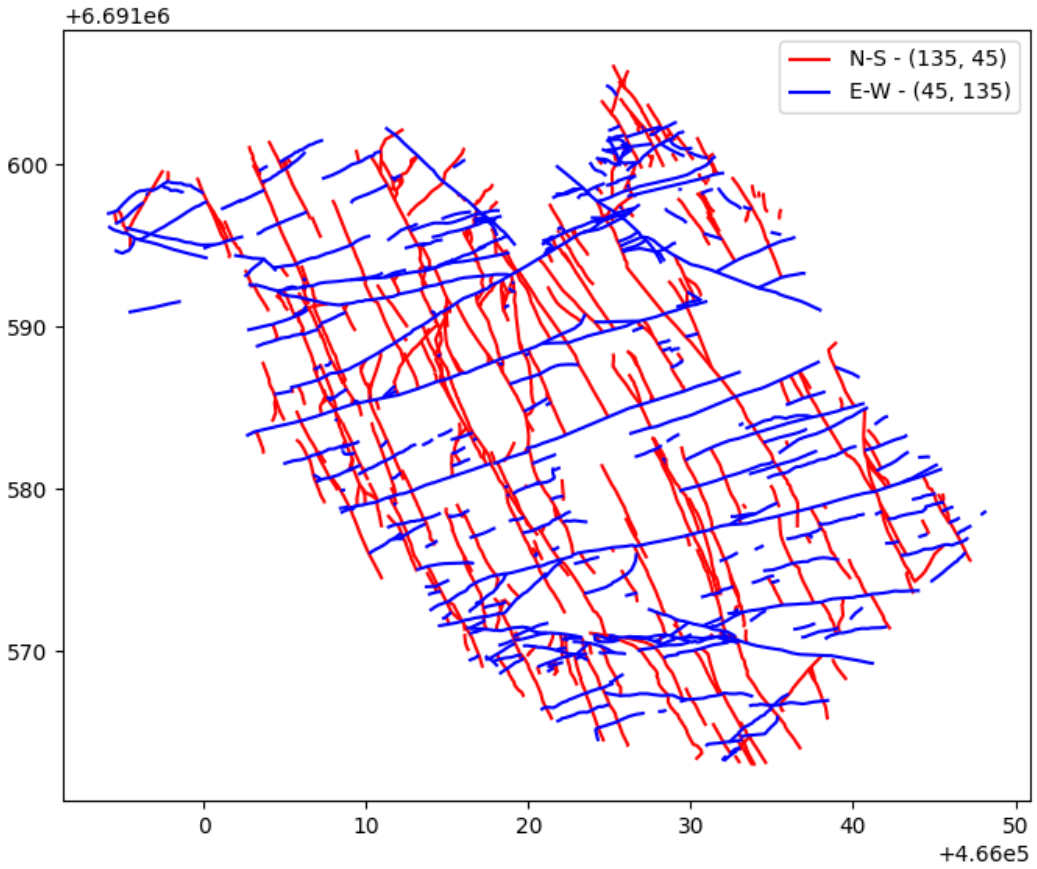
```
pprint((kb11_network.azimuth_set_names, kb11_network.azimuth_set_ranges))
```

```
(('N-S', 'E-W'), ((135, 45), (45, 135)))
```

```
pprint(kb11_network.trace_azimuth_set_counts)
```

```
{'E-W': 315, 'N-S': 394}
```

```
fig, ax = plt.subplots(figsize=(8, 8))
colors = ("red", "blue")
assert len(colors) == len(kb11_network.azimuth_set_names)
for azimuth_set, set_range, color in zip(
    kb11_network.azimuth_set_names, kb11_network.azimuth_set_ranges, colors
):
    trace_gdf_set = kb11_network.trace_gdf.loc[
        kb11_network.trace_gdf["azimuth_set"] == azimuth_set
    ]

    trace_gdf_set.plot(color=color, label=f"{azimuth_set} - {set_range}", ax=ax)
plt.legend()
plt.show()
```



Total running time of the script: (0 minutes 0.283 seconds)

Plotting the trace data used as input in `fractopo`

Data is loaded into `fractopo` using `geopandas` which can load from a wide variety of sources.

Here we load data from the internet from the `fractopo` GitHub repository.

```
# Import geopandas for loading and handling gis data
import geopandas as gpd

# Import matplotlib for plotting
import matplotlib.pyplot as plt
```

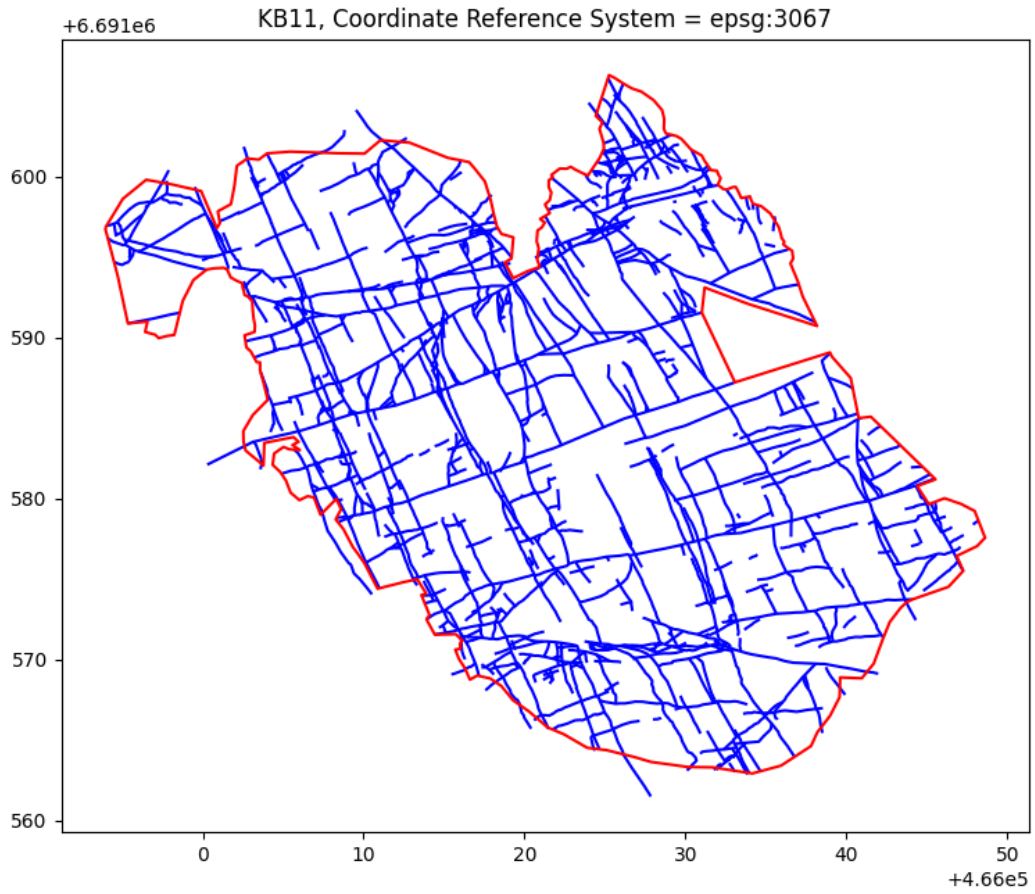
(continues on next page)

(continued from previous page)

```
trace_data_url = (
    "https://raw.githubusercontent.com/nialov/"
    "fractopo/master/tests/sample_data/KB11/KB11_traces.geojson"
)
area_data_url = (
    "https://raw.githubusercontent.com/nialov/"
    "fractopo/master/tests/sample_data/KB11/KB11_area.geojson"
)

# Use geopandas to load data from urls
traces = gpd.read_file(trace_data_url)
area = gpd.read_file(area_data_url)

# Check that the type is GeoDataFrame
assert isinstance(traces, gpd.GeoDataFrame)
assert isinstance(area, gpd.GeoDataFrame)

# Name the dataset
name = "KB11"
```

Plotting the loaded data

```
# Initialize the figure and ax in which data is plotted
fig, ax = plt.subplots(figsize=(9, 9))

# Plot the loaded trace dataset consisting of fracture traces.
traces.plot(ax=ax, color="blue")

# Plot the loaded area dataset that consists of a single polygon
# that delineates the traces.
area.boundary.plot(ax=ax, color="red")

# Give the figure a title
ax.set_title(f"{name}, Coordinate Reference System = {traces.crs}")
```




```
Text(0.5, 1.0, 'KB11, Coordinate Reference System = epsg:3067')
```

Total running time of the script: (0 minutes 0.504 seconds)

Plotting length distributions with `fractopo`

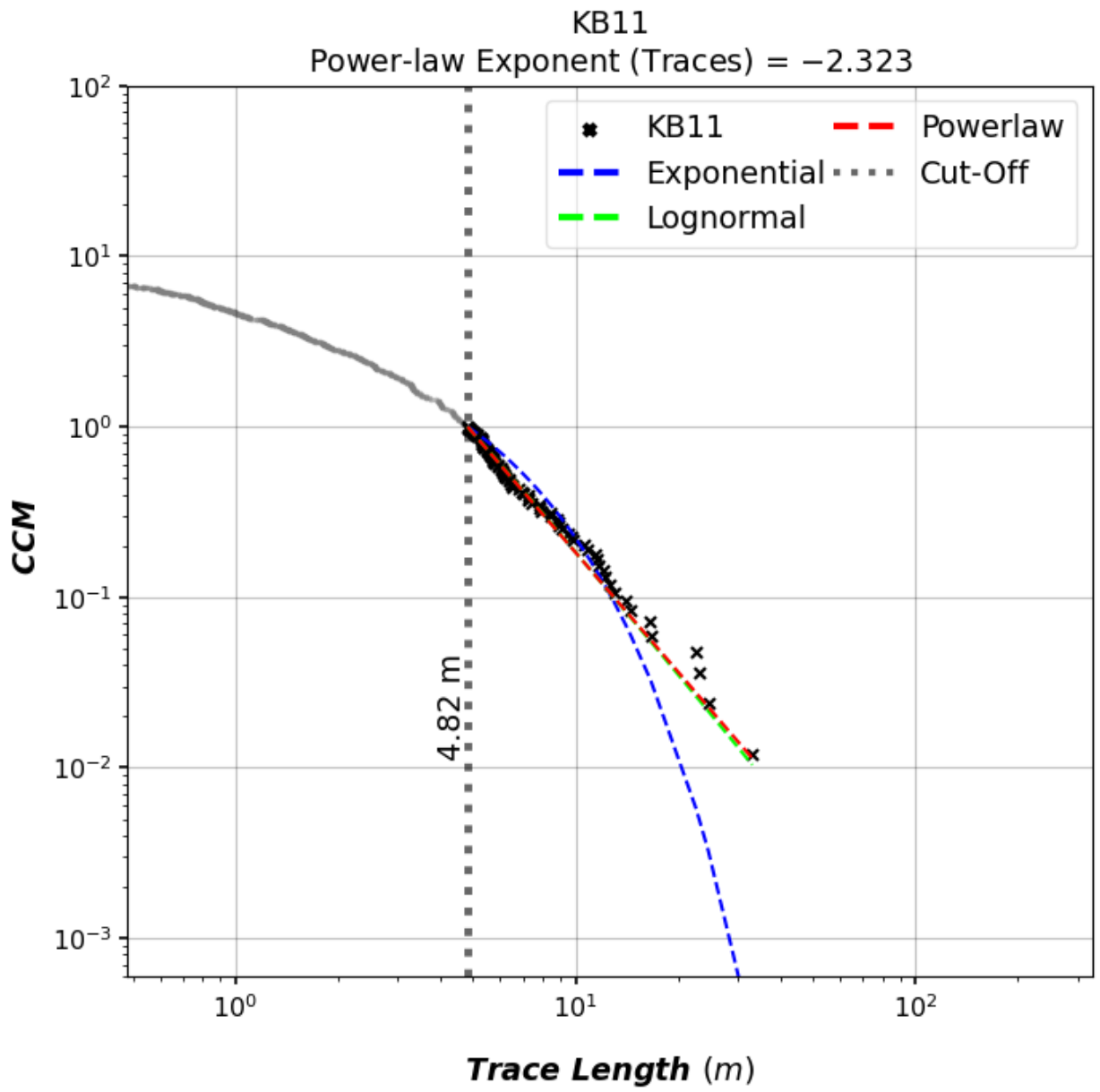
Initializing

```
from pprint import pprint

import matplotlib as mpl
import matplotlib.pyplot as plt

# Load kb11_network network from examples/example_networks.py
from example_networks import kb11_network

mpl.rcParams["figure.figsize"] = (5, 5)
mpl.rcParams["font.size"] = 8
```

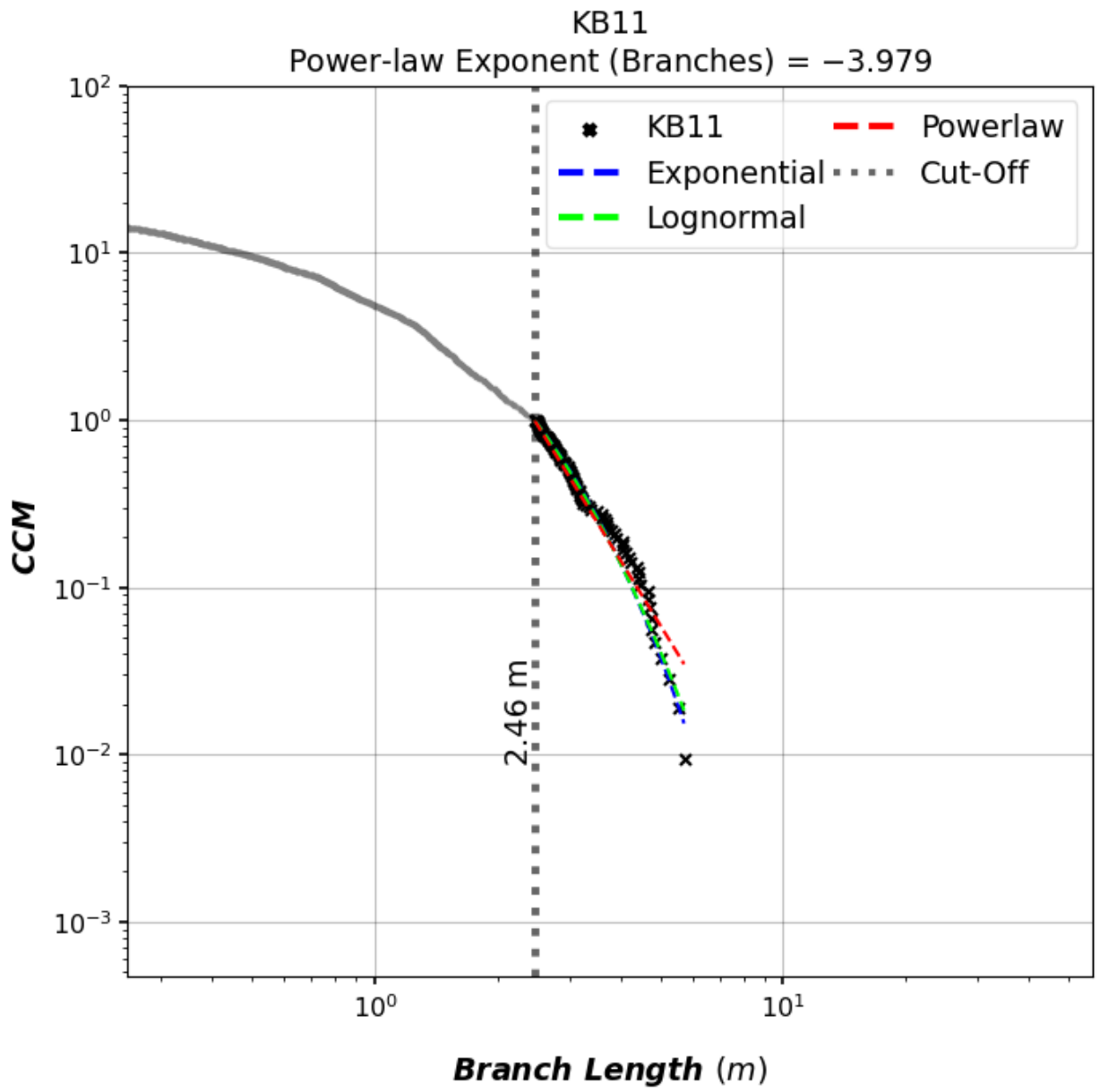
Plotting length distribution plots of fracture traces and branches

Using Complementary Cumulative Number/Function

```
# Log-log plot of network trace length distribution
fit, fig, ax = kb11_network.plot_trace_lengths()

# Use matplotlib helpers to make sure plot fits in the gallery webpage!
# (Not required.)
plt.tight_layout()
plt.show()
```

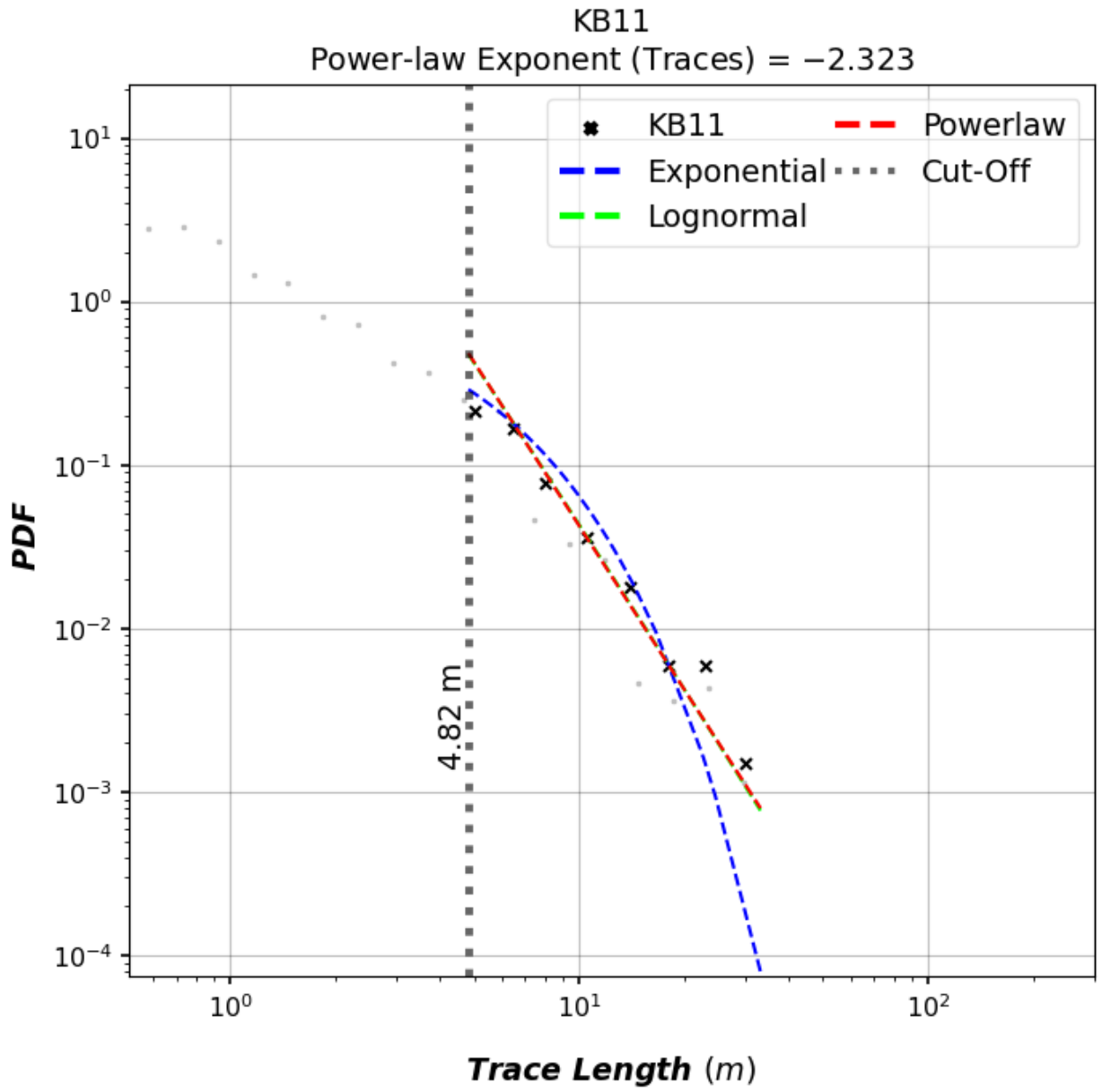



```
# Log-log plot of network branch length distribution
kb11_network.plot_branch_lengths()
plt.tight_layout()
plt.show()
```

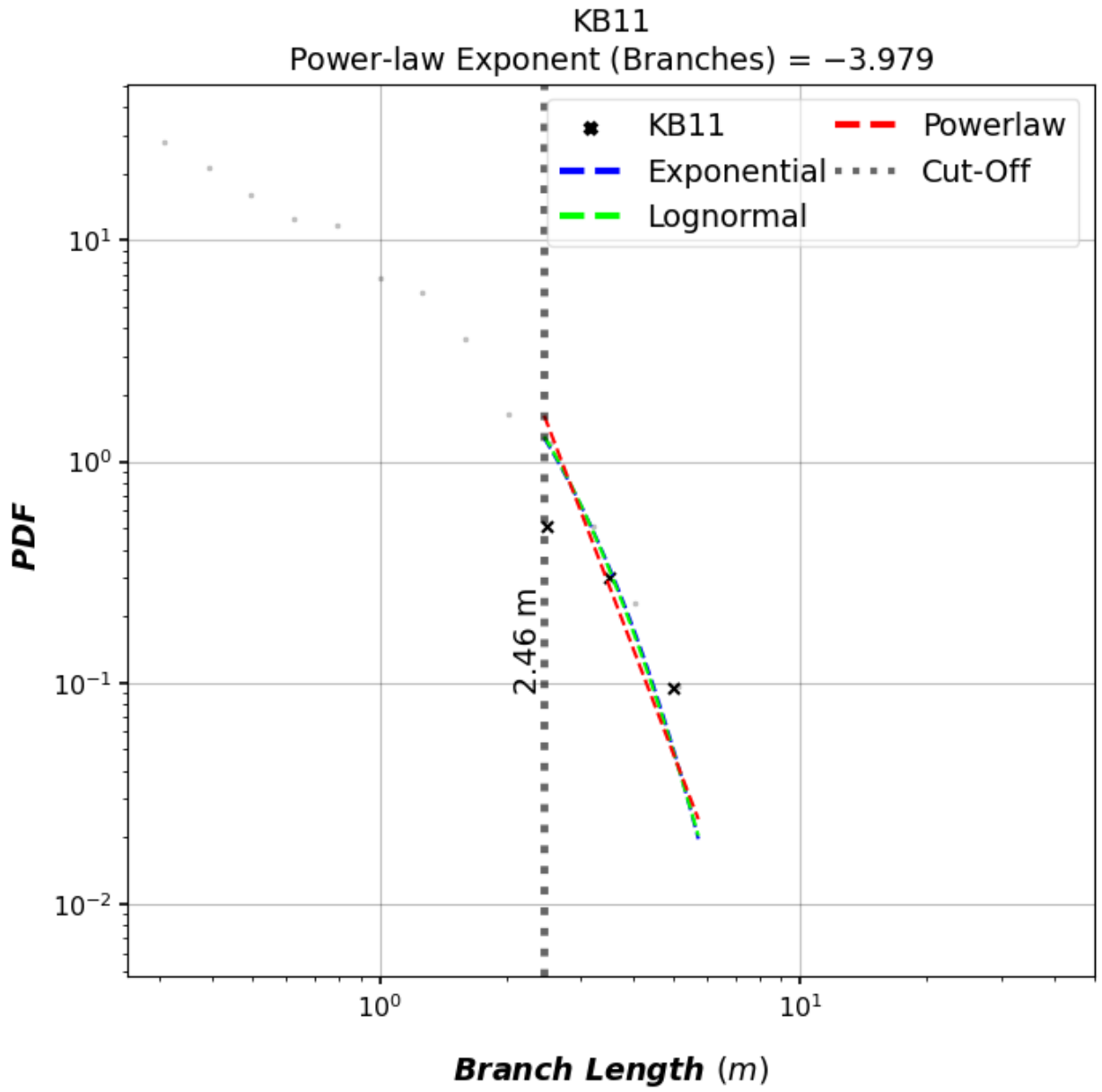


Using Probability Density Function

```
# Log-log plot of network trace length distribution
kb11_network.plot_trace_lengths(use_probability_density_function=True)

# Use matplotlib helpers to make sure plot fits in the gallery webpage!
# (Not required.)
plt.tight_layout()
plt.show()
```




```
# Log-log plot of network branch length distribution
kb11_network.plot_branch_lengths(use_probability_density_function=True)
plt.tight_layout()
plt.show()
```



Numerical descriptions of fits are accessible as properties

```
# Use pprint for printing with prettier output
pprint(kb11_network.trace_lengths_powerlaw_fit_description)
```

```
{'trace exponential Kolmogorov-Smirnov distance D': 0.16734893835952314,
 'trace exponential lambda': 0.2898919999912063,
 'trace exponential loglikelihood': -188.02077841990496,
 'trace lengths cut off proportion': 0.8815232722143864,
 'trace lognormal Kolmogorov-Smirnov distance D': 0.055285689798228566,
 'trace lognormal loglikelihood': -181.36559959113393,
```

(continues on next page)

(continued from previous page)

```
 'trace lognormal mu': -24.654367132184213,
 'trace lognormal sigma': 3.413632605089442,
 'trace lognormal vs. exponential R': 1.840247791139716,
 'trace lognormal vs. exponential p': 0.06573186649520334,
 'trace power_law Kolmogorov-Smirnov distance D': 0.05261149958762154,
 'trace power_law alpha': 3.323399316187791,
 'trace power_law cut-off': 4.815579557192599,
 'trace power_law exponent': -2.323399316187791,
 'trace power_law sigma': 0.25350364847712353,
 'trace power_law vs. exponential R': 1.792390864166387,
 'trace power_law vs. exponential p': 0.07307037719921719,
 'trace power_law vs. lognormal R': -0.09973967423028064,
 'trace power_law vs. lognormal p': 0.9205510020886868,
 'trace power_law vs. truncated_power_law R': -0.3949163478172287,
 'trace power_law vs. truncated_power_law p': 0.6549247892396883,
 'trace truncated_power_law Kolmogorov-Smirnov distance D': 0.06088951719124519,
 'trace truncated_power_law alpha': 3.101655682133221,
 'trace truncated_power_law exponent': -2.101655682133221,
 'trace truncated_power_law lambda': 0.01693723427355317,
 'trace truncated_power_law loglikelihood': -181.27535493663555}
```

```
pprint(kb11_network.branch_lengths_powerlaw_fit_description)
```

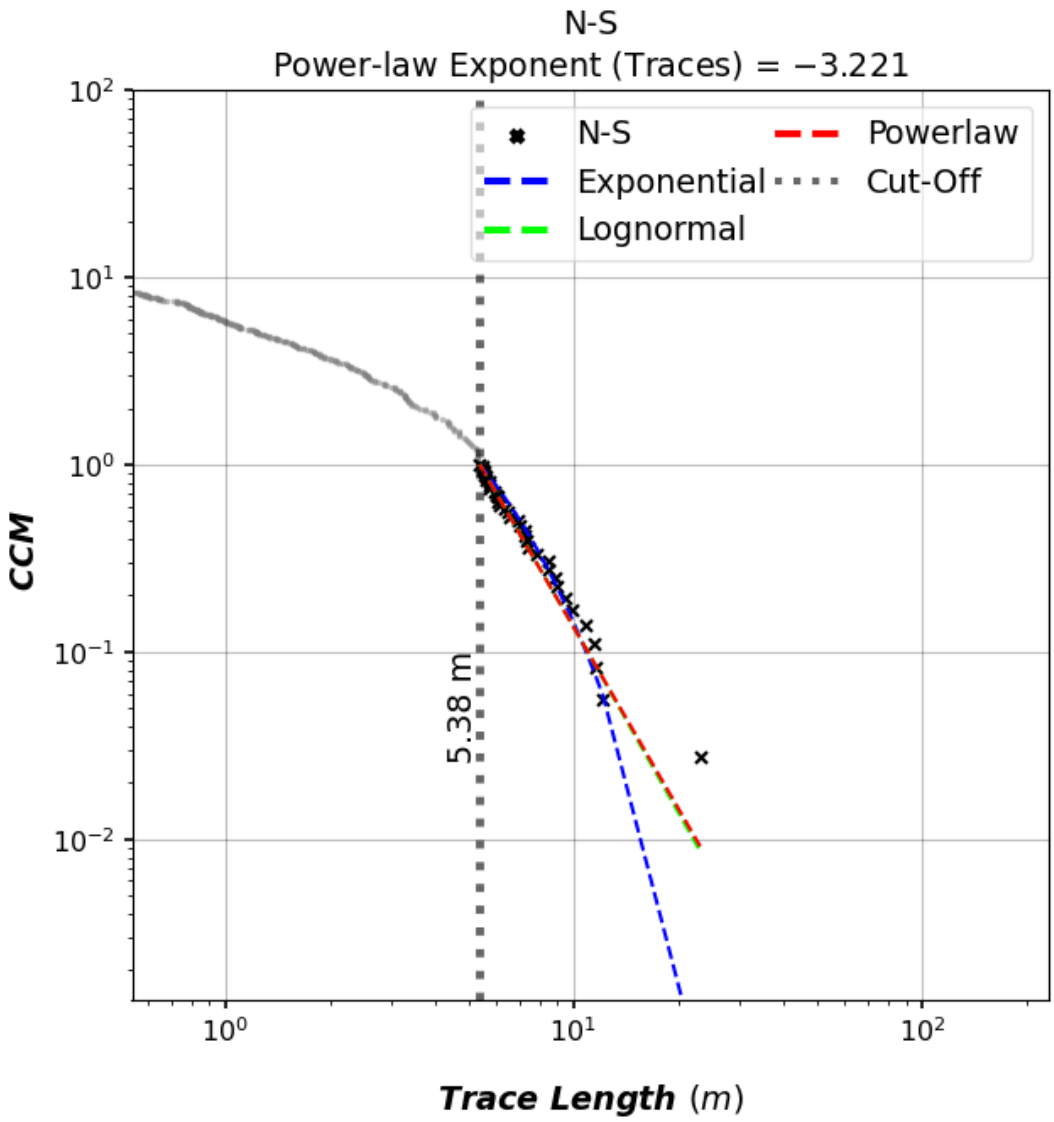
```
{'branch exponential Kolmogorov-Smirnov distance D': 0.055610337970091184,
 'branch exponential lambda': 1.2799623045206825,
 'branch exponential loglikelihood': -79.83654595014787,
 'branch lengths cut off proportion': 0.9488416988416989,
 'branch lognormal Kolmogorov-Smirnov distance D': 0.056242712153426244,
 'branch lognormal loglikelihood': -80.11030644664197,
 'branch lognormal mu': 0.3954002513747114,
 'branch lognormal sigma': 0.48573668994587976,
 'branch lognormal vs. exponential R': -1.2860976878834365,
 'branch lognormal vs. exponential p': 0.1984089709536515,
 'branch power_law Kolmogorov-Smirnov distance D': 0.05150595395106594,
 'branch power_law alpha': 4.9790958462418935,
 'branch power_law cut-off': 2.4602426467400713,
 'branch power_law exponent': -3.9790958462418935,
 'branch power_law sigma': 0.38648395404192654,
 'branch power_law vs. exponential R': -1.2796975009064637,
 'branch power_law vs. exponential p': 0.20065154388298556,
 'branch power_law vs. lognormal R': -1.2484090815914064,
 'branch power_law vs. lognormal p': 0.21188128491687686,
 'branch power_law vs. truncated_power_law R': -1.5645011856154882,
 'branch power_law vs. truncated_power_law p': 0.055258061048291784,
 'branch truncated_power_law Kolmogorov-Smirnov distance D': 0.05467641228173603,
 'branch truncated_power_law alpha': 1.0000206602209936,
 'branch truncated_power_law exponent': -2.066022099356246e-05,
 'branch truncated_power_law lambda': 1.0171370451351636,
 'branch truncated_power_law loglikelihood': -79.83784521401718}
```

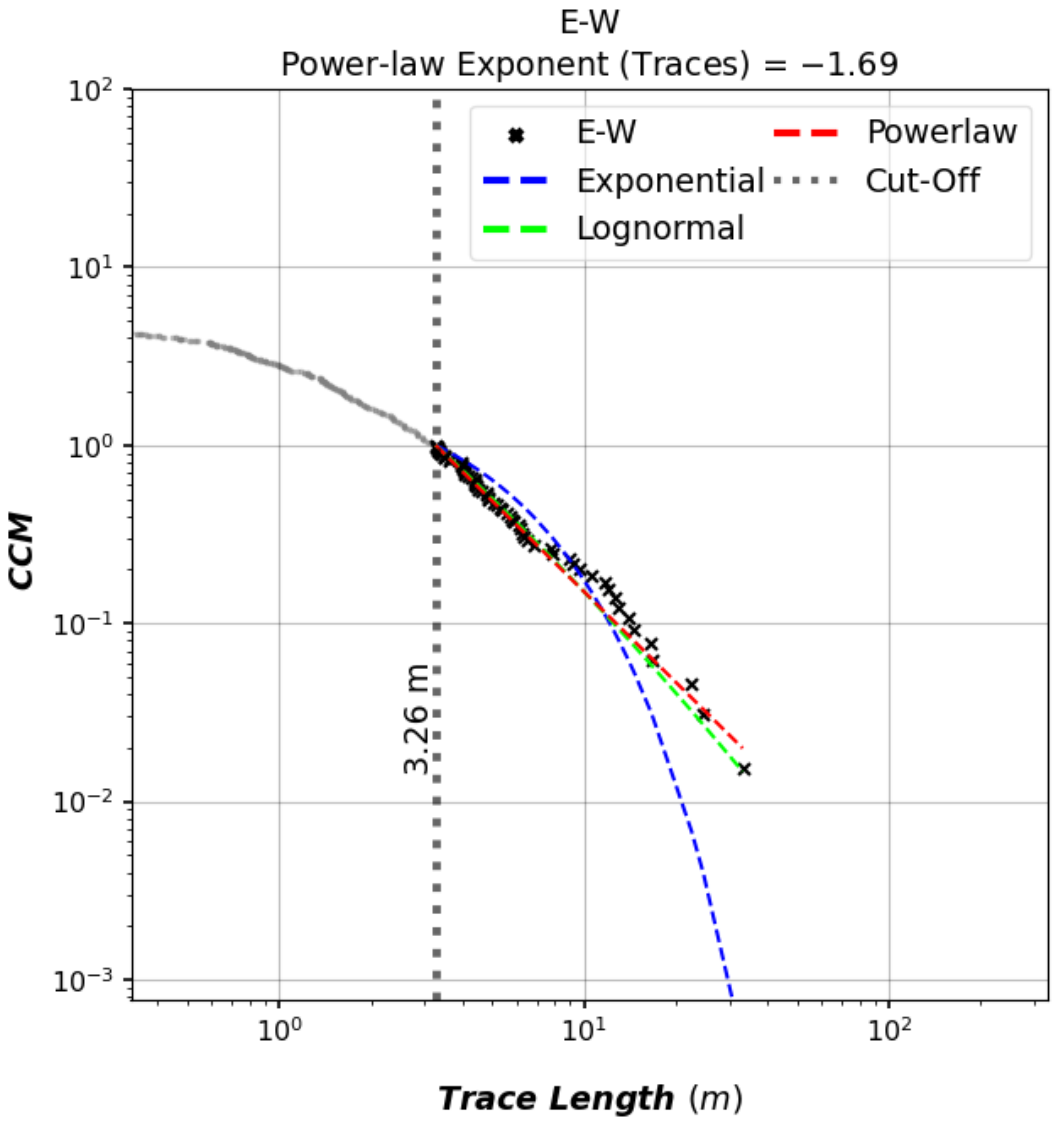
Set-wise length distribution plotting

```
pprint(kb11_network.azimuth_set_names)
pprint(kb11_network.azimuth_set_ranges)
```

```
('N-S', 'E-W')
((135, 45), (45, 135))
```

```
fits, figs, axes = kb11_network.plot_trace_azimuth_set_lengths()
```





Total running time of the script: (0 minutes 3.536 seconds)

Numerical network characteristics

Lineament & fracture networks can be characterized with numerous geometric and topological parameters.

Initializing

Create two Networks so that their numerical attributes can be compared.

```
from pprint import pprint

import geopandas as gpd
import matplotlib.pyplot as plt

# Load kb11_network network from examples/example_networks.py
from example_networks import kb11_network

# Import Network class from fractopo
from fractopo import Network
from fractopo.analysis.parameters import plot_parameters_plot

# Make kb7_network here
kb7_network = Network(
    name="KB7",
    trace_gdf=gpd.read_file(
        "https://raw.githubusercontent.com/nialov/"
        "fractopo/master/tests/sample_data/KB7/KB7_traces.geojson"
    ),
    area_gdf=gpd.read_file(
        "https://raw.githubusercontent.com/nialov/"
        "fractopo/master/tests/sample_data/KB7/KB7_area.geojson"
    ),
    truncate_traces=True,
    circular_target_area=False,
    determine_branches_nodes=True,
    snap_threshold=0.001,
)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/fractopo/envs/stable/lib/python3.8/site-
↪packages/geopandas/tools/clip.py:67: FutureWarning: In a future version, `df.iloc[:,␣
↪i] = newvals` will attempt to set the values inplace instead of always setting a new␣
↪array. To retain the old behavior, use either `df[df.columns[i]] = newvals` or, if␣
↪columns are non-unique, `df.isetitem(i, newvals)`
  clipped.loc[
Point already matches a coordinate point in trace.
point: POINT (466024.01981982775 6692098.386310579), trace: LINESTRING (466023.
↪42354138196 6692098.175793974, 466023.56934693456 6692098.227270745, 466023.
↪71742809657 6692098.279550923, 466024.01981982775 6692098.386310579, 466024.2524532033␣
↪6692098.468441982, 466024.55418436136 6692098.572371604, 466025.4048296623 6692098.
↪865371656, 466025.64345226996 6692098.9475638885, 466025.68120606244 6692098.96056797,␣
↪466027.39570949227 6692099.43681892, 466028.1894610794 6692099.627319304)
Returning original trace without changes.
```

Geometric and topological Network parameters

All parameters are accessible through an attribute

```
kb11_parameters = kb11_network.parameters
kb7_parameters = kb7_network.parameters
```

```
pprint(kb11_parameters)
```

```
{'Area': 1237.9003657531266,
 'Areal Frequency B20': 1.627756203766927,
 'Areal Frequency P20': 0.5258904658323919,
 'Branch Max Length': 5.721558550556387,
 'Branch Mean Length': 0.7761437911315212,
 'Branch Min Length': 0.006639501944314653,
 'Connection Frequency': 0.8837544848243267,
 'Connections per Branch': 1.7627791563275435,
 'Connections per Trace': 3.3609831029185866,
 'Dimensionless Intensity B22': 0.9805590097335628,
 'Dimensionless Intensity P22': 2.786779132035443,
 'Fracture Density (Mauldon)': nan,
 'Fracture Intensity (Mauldon)': nan,
 'Fracture Intensity B21': 1.2633728710295158,
 'Fracture Intensity P21': 1.2633728710295158,
 'Number of Branches': 2015.0,
 'Number of Branches (Real)': 2072,
 'Number of Traces': 651.0,
 'Number of Traces (Real)': 709,
 'Trace Max Length': 33.1085306416674,
 'Trace Mean Length': 2.2058247378420526,
 'Trace Mean Length (Mauldon)': nan,
 'Trace Min Length': 5.820766091346741e-10}
```

```
pprint(kb7_parameters)
```

```
{'Area': 534.6848339727045,
 'Areal Frequency B20': 0.7209854768756723,
 'Areal Frequency P20': 0.37685751904136955,
 'Branch Max Length': 11.141338433454226,
 'Branch Mean Length': 1.1849926370920998,
 'Branch Min Length': 0.002572582127554804,
 'Connection Frequency': 0.26183648965653467,
 'Connections per Branch': 1.2036316472114137,
 'Connections per Trace': 1.3895781637717122,
 'Dimensionless Intensity B22': 1.0124132500421243,
 'Dimensionless Intensity P22': 1.6261887828801624,
 'Fracture Density (Mauldon)': nan,
 'Fracture Intensity (Mauldon)': nan,
 'Fracture Intensity B21': 0.854362481548008,
 'Fracture Intensity P21': 0.854362481548008,
 'Number of Branches': 385.5,
 'Number of Branches (Real)': 423,
```

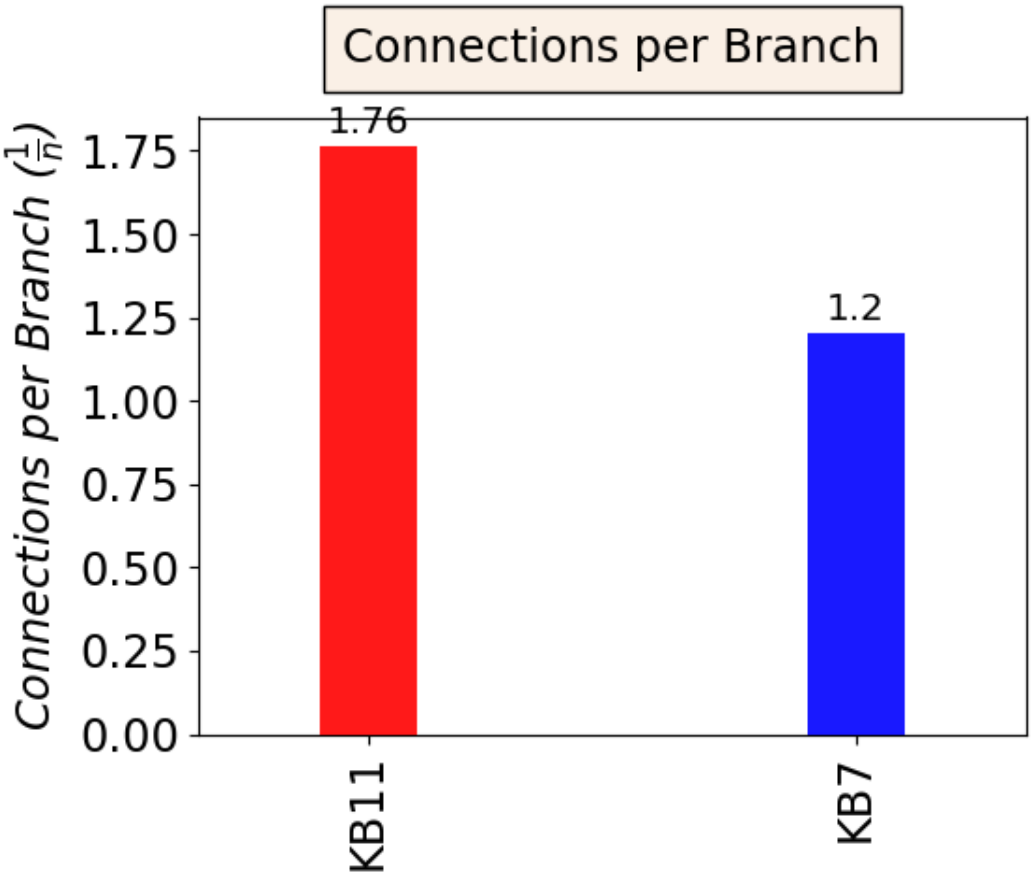
(continues on next page)

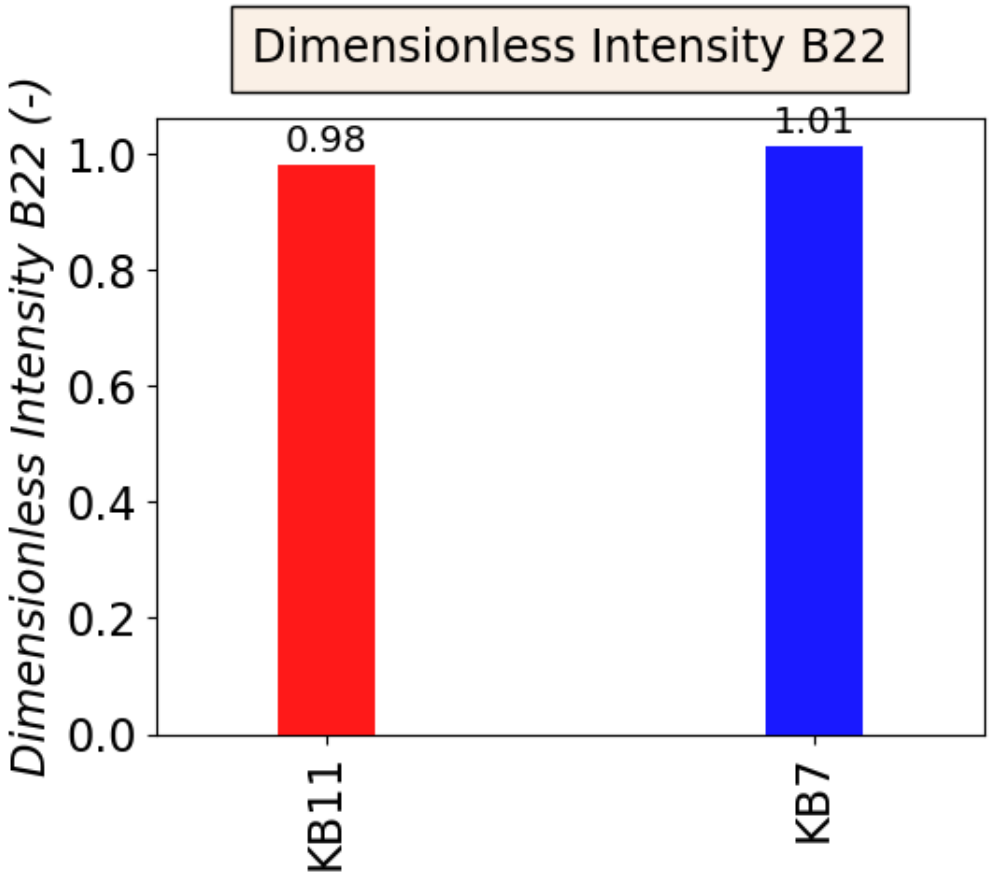
(continued from previous page)

```
 'Number of Traces': 201.5,
 'Number of Traces (Real)': 240,
 'Trace Max Length': 11.943520313323264,
 'Trace Mean Length': 1.9033944233291853,
 'Trace Mean Length (Mauldon)': nan,
 'Trace Min Length': 0.06136251196974645}
```

Compare KB11 and KB7 Networks selected parameter values

```
b22 = "Dimensionless Intensity B22"
cpb = "Connections per Branch"
selected = {b22, cpb}

# Filter to only selected parameters
kb11_network_selected_params = {
    param: value for param, value in kb11_parameters.items() if param in selected
}
kb7_network_selected_params = {
    param: value for param, value in kb7_parameters.items() if param in selected
}

# Compare parameters with a simple bar plot
figs, axes = plot_parameters_plot(
    topology_parameters_list=[
        kb11_network_selected_params,
        kb7_network_selected_params,
    ],
    labels=["KB11", "KB7"],
    colors=["red", "blue"],
)
plt.show()
```


Connections per Branch
Connections per Branch ($\frac{1}{n}$)
1.76
1.2
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
KB11
KB7



-

Total running time of the script: (0 minutes 1.707 seconds)

Plotting multi-scale fracture networks with `fractopo`

Initializing

```
import matplotlib as mpl
import matplotlib.pyplot as plt

# Load kb11_network and hastholmen_network
from example_networks import hastholmen_network, kb11_network

from fractopo import MultiNetwork

mpl.rcParams["figure.figsize"] = (5, 5)
mpl.rcParams["font.size"] = 8
```

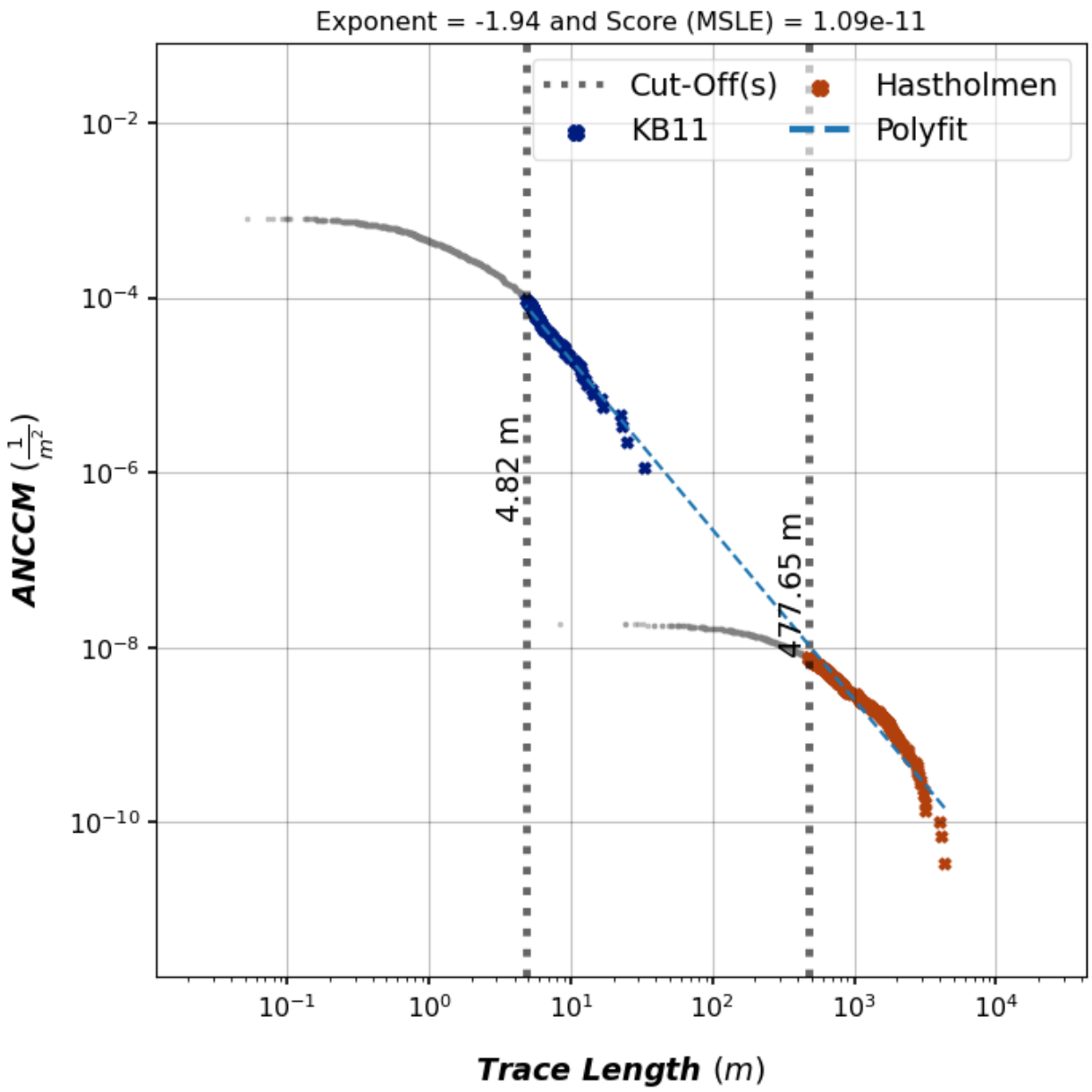
Create MultiNetwork object

```
multi_network = MultiNetwork((kb11_network, hastholmen_network))
```

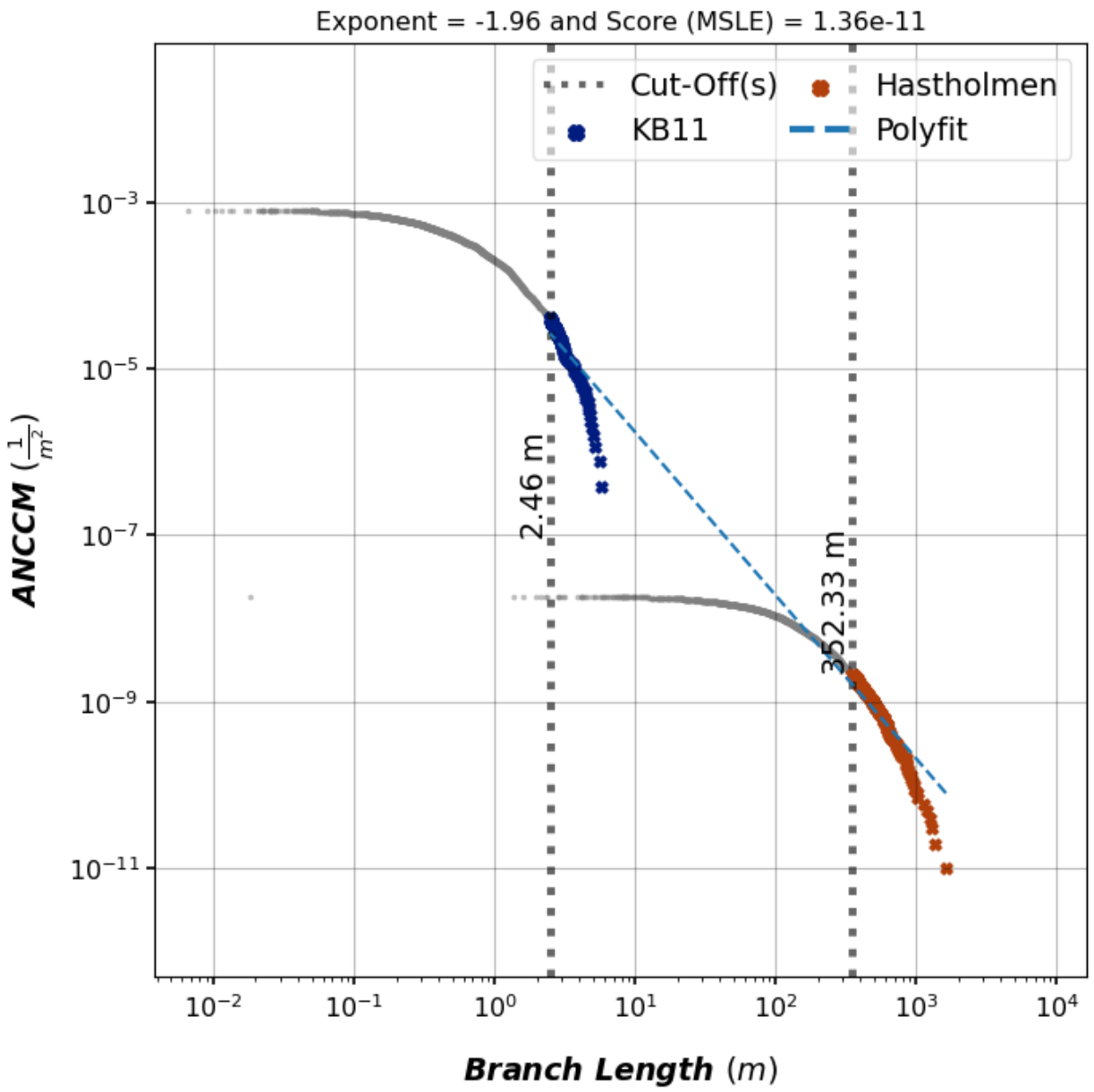
Plot automatically cut length distributions with a multi-scale fit

Powerlaw exponent and fit 'score' are embedded into the plots. MSLE = Mean Squared Logarithmic Error

```
# Log-log plot of MultiNetwork trace length distribution
mld_traces, polyfit, fig, ax = multi_network.plot_multi_length_distribution(
    using_branches=False,
    automatic_cut_offs=True,
    plot_truncated_data=True,
)

# Visual plot setup
plt.tight_layout()
```




```
# Log-log plot of MultiNetwork branch length distribution
mld_branches, polyfit, fig, ax = multi_network.plot_multi_length_distribution(
    using_branches=True,
    automatic_cut_offs=True,
    plot_truncated_data=True,
)

# Visual plot setup
plt.tight_layout()
```



Numerical details of multi-scale length distributions

```
# The returned MultiLengthDistribution objects contain details
print(f"Exponent of traces fit: {polyfit.m_value}")
```

```
Exponent of traces fit: -1.9628074179036543
```

```
print(f"Exponent of branches fit: {polyfit.m_value}")
```

```
Exponent of branches fit: -1.9628074179036543
```

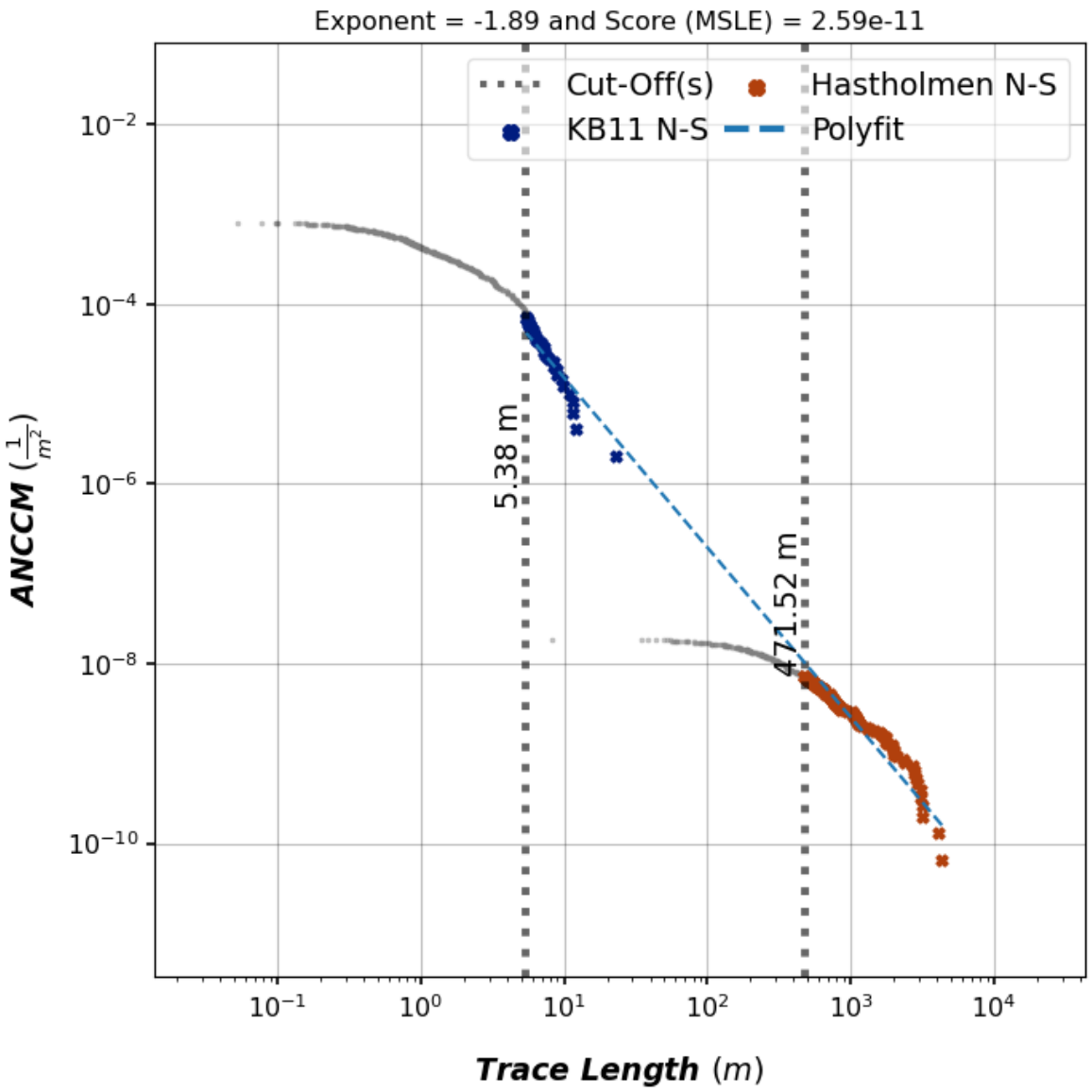
Plot set-wise multi-scale distributions for traces

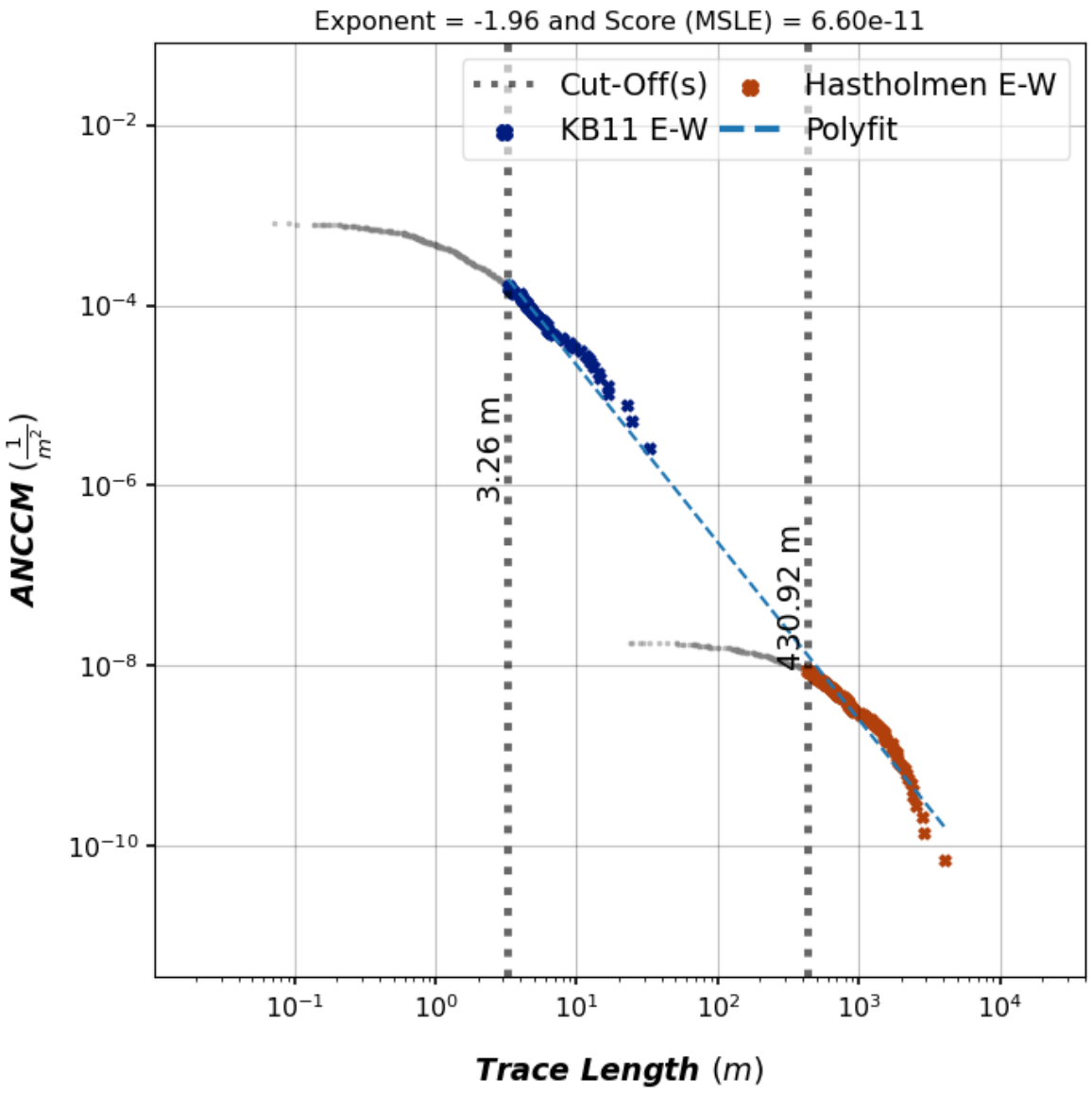
Requires that same azimuth sets are defined in all Networks in the MultiNetwork.

```
# Set names and ranges
# MultiNetwork.collective_azimuth_sets() will check that same azimuth sets are
# set in all Networks.
print(multi_network.collective_azimuth_sets())
```

```
(('N-S', 'E-W'), ((135, 45), (45, 135)))
```

```
mlds, polyfits, figs, axes = multi_network.plot_trace_azimuth_set_lengths(
    automatic_cut_offs=True,
    plot_truncated_data=True,
)

# Just some visual plot setup...
for fig in figs:
    fig.tight_layout()
```


Exponent = -1.89 and Score (MSLE) = 2.59e-11
Cut-Off(s)
Hastholmen N-S
KB11 N-S
Polyfit
5.38 m
471.52 m
ANCCM ($\frac{1}{m^2}$)
Trace Length (m)

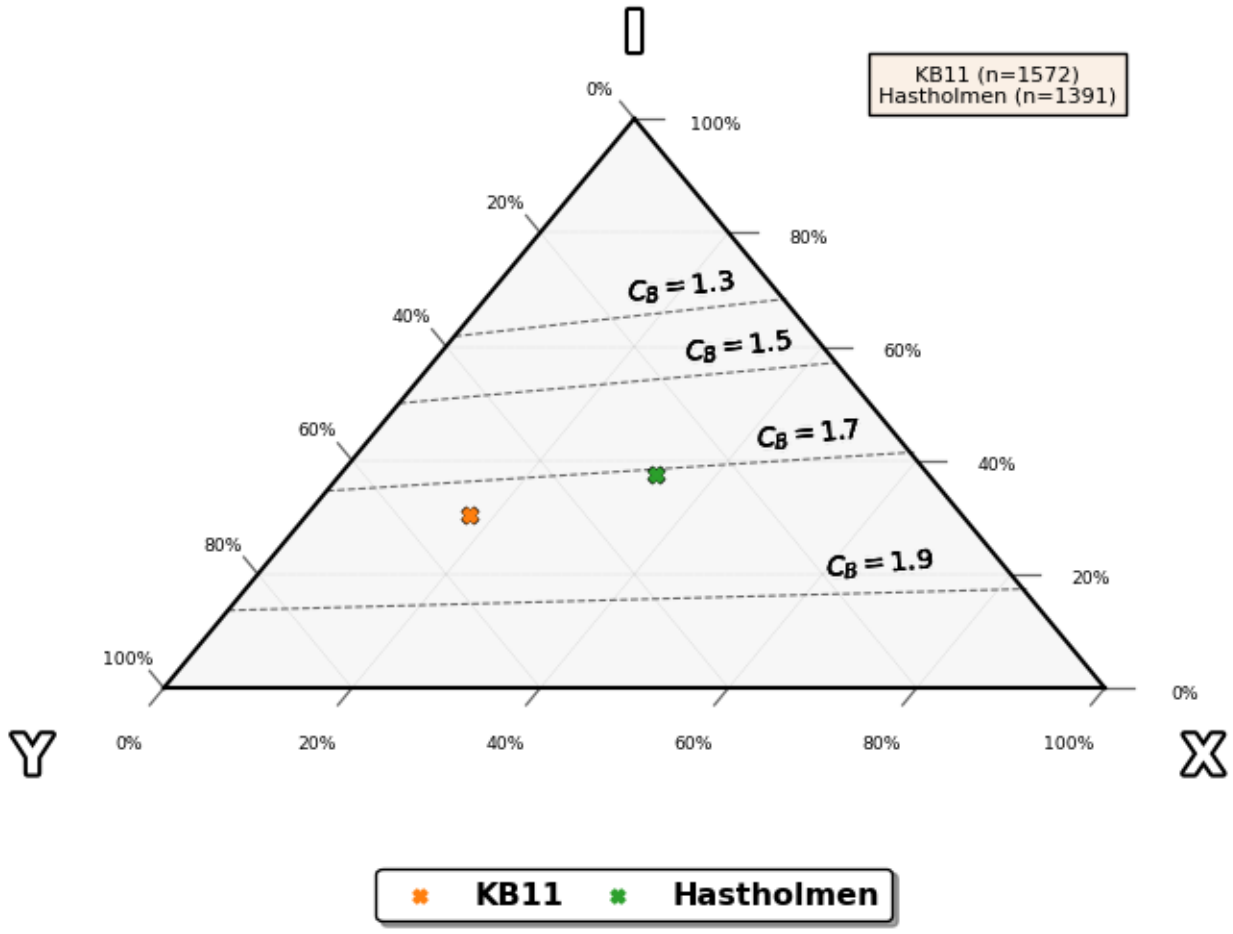


-

Plot ternary plots of nodes and branches

Topological XYI-node plot

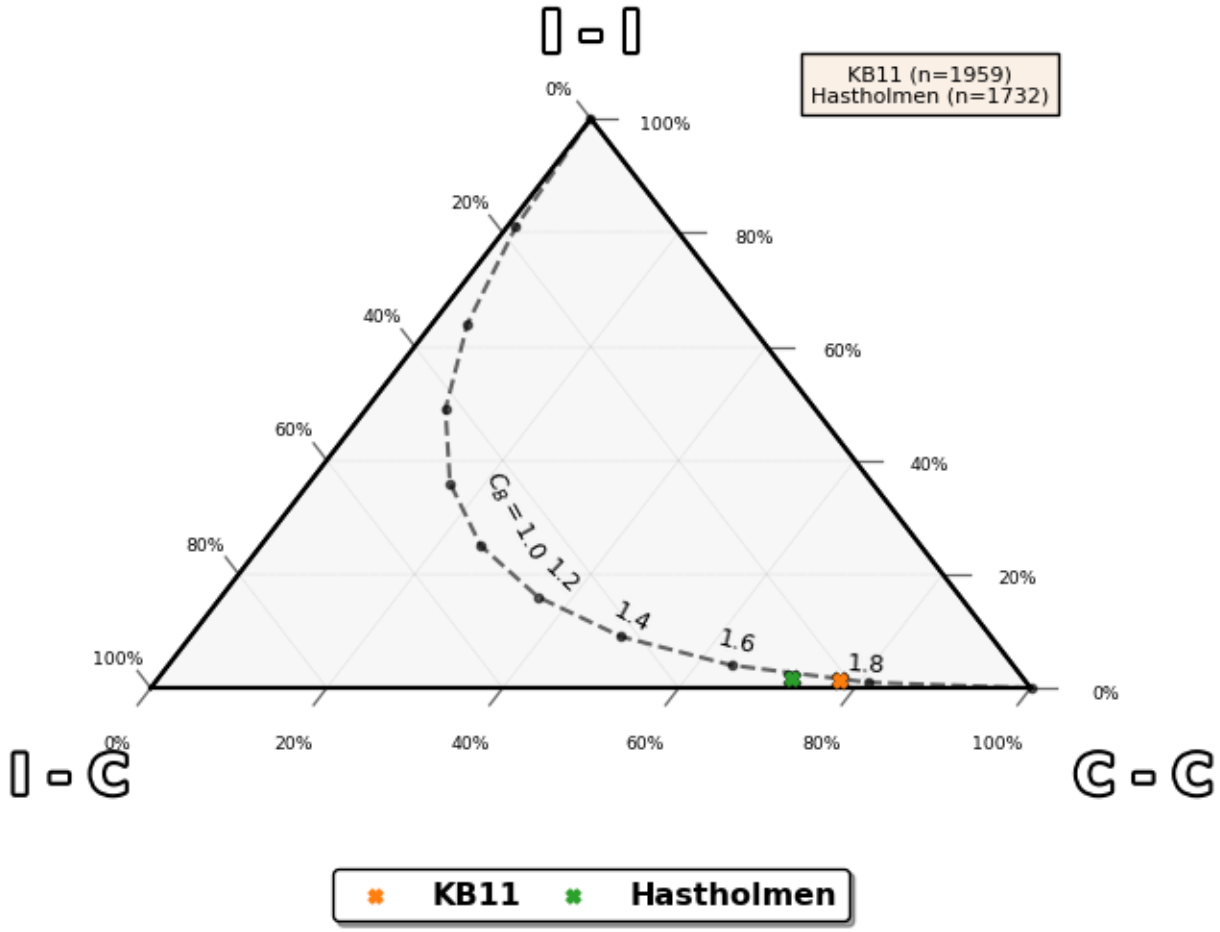
```
fig, ax, tax = multi_network.plot_xyi()

# Visual plot setup
plt.tight_layout()
```




```
/home/docs/checkouts/readthedocs.org/user_builds/fractopo/envs/stable/lib/python3.8/site-
↪packages/ternary/plotting.py:148: UserWarning: No data for colormapping provided via 'c
↪'. Parameters 'vmin', 'vmax' will be ignored
  ax.scatter(xs, ys, vmin=vmin, vmax=vmax, **kwargs)
```

Topological branch type plot

```
fig, ax, tax = multi_network.plot_branch()

# Visual plot setup
plt.tight_layout()
```




```
/home/docs/checkouts/readthedocs.org/user_builds/fractopo/envs/stable/lib/python3.8/site-
↪packages/ternary/plotting.py:148: UserWarning: No data for colormapping provided via 'c
↪'. Parameters 'vmin', 'vmax' will be ignored
  ax.scatter(xs, ys, vmin=vmin, vmax=vmax, **kwargs)
```

Total running time of the script: (0 minutes 4.768 seconds)

Determining topological branches and nodes

A network consists of the geometrical traces and their interactions with each other.

Imports

```
import geopandas as gpd
import matplotlib.pyplot as plt

# Import the geometries used to create traces and target areas.
from shapely.geometry import LineString, Polygon

# Function to determine branches and nodes
```

(continues on next page)

(continued from previous page)

```
# Network, when initialized with determine_branches_nodes=True,
# will call this to determine them internally.
from fractopo.branches_and_nodes import branches_and_nodes
```

Define trace and target area geometries manually

```
traces = gpd.GeoDataFrame(
    {
        "geometry": [
            LineString([(-2, 0), (4, 0)]),
            LineString([(0, -2), (0, 4)]),
            LineString([(-1, 1), (0, 1)]),
        ]
    }
)

area = gpd.GeoDataFrame({"geometry": [Polygon([(-2, -2), (2, -2), (2, 2), (-2, 2)])]})
```

Plot the traces and target area with their branches and nodes

After we've manually created some traces and delineated their target area with the `area` `Polygon` we can determine branches and nodes of the traces network.

You may notice that the branches and nodes are cropped to the original target area. Branches and nodes will never be determined outside the target area.

Determine branches and nodes

```
branches, nodes = branches_and_nodes(traces, area, snap_threshold=0.001)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/fractopo/envs/stable/lib/python3.8/site-
↪packages/pygeos/io.py:89: UserWarning: The shapely GEOS version (3.10.3-CAPI-1.16.1)␣
↪is incompatible with the PyGEOS GEOS version (3.10.4-CAPI-1.16.2). Conversions between␣
↪both will be slow
  warnings.warn(
```

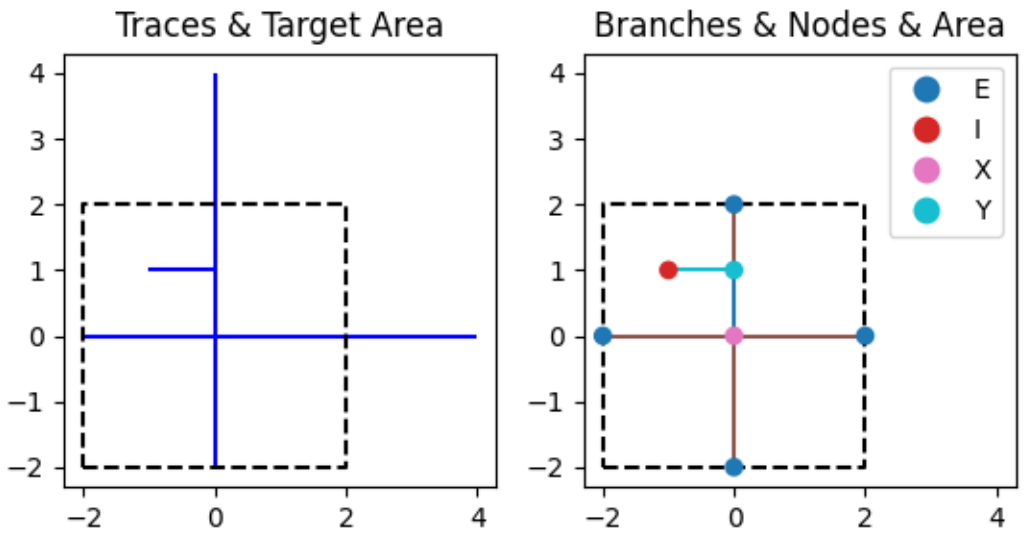
Plot the data

```
# Initialize matplotlib figure and two axes
# One axis is for traces and other for determined branches and nodes
fig, axes = plt.subplots(1, 2)

# Plot traces
traces.plot(ax=axes[0], color="blue", label="Traces")

# Plot the area boundary, not the full Polygon
```

(continues on next page)

(continued from previous page)

```
area.boundary.plot(ax=axes[0], color="black", label="Target Area", linestyle="dashed")
axes[0].set_title("Traces & Target Area")

# Plot the created branches and nodes
branches_axes = branches.plot(ax=axes[1], column="Connection", categorical=True)
nodes.plot(ax=axes[1], column="Class", zorder=10, legend=True, categorical=True)
axes[1].set_title("Branches & Nodes & Area")

# Plot the area boundary to the other ax as well
area.boundary.plot(ax=axes[1], color="black", linestyle="dashed")

# Set second plot boundaries to same as first
x_min, y_min, x_max, y_max = traces.total_bounds
axes[1].set_xlim(*axes[0].get_xlim())
axes[1].set_ylim(*axes[0].get_ylim())

# Show the plot
plt.show()
```



Total running time of the script: (0 minutes 0.515 seconds)

Optimizing multi-scale cut-offs with `fractopo`

This functionality is very much so a **work-in-progress**. Optimization of a power-law fit for continuous, "single-scale", data is easily handled by the functionality provided by `powerlaw` but it does not directly translate to the required methods for multi-scale data where normalization of the complementary cumulative number (=ccm) has been done.

Initializing

```
import matplotlib as mpl
import matplotlib.pyplot as plt

# Load three networks, each digitized from a different scale of observation
from example_networks import hastholmen_network, kb11_network, lidar_200k_network

from fractopo import general
from fractopo.analysis import length_distributions

mpl.rcParams["figure.figsize"] = (5, 5)
mpl.rcParams["font.size"] = 8
```

Collect LengthDistributions into MultiLengthDistribution

```
networks = [kb11_network, hastholmen_network, lidar_200k_network]

distributions = [netw.trace_length_distribution(azimuth_set=None) for netw in networks]

mld = length_distributions.MultiLengthDistribution(
    distributions=distributions,
    using_branches=False,
    fitter=length_distributions.scikit_linear_regression,
)
```

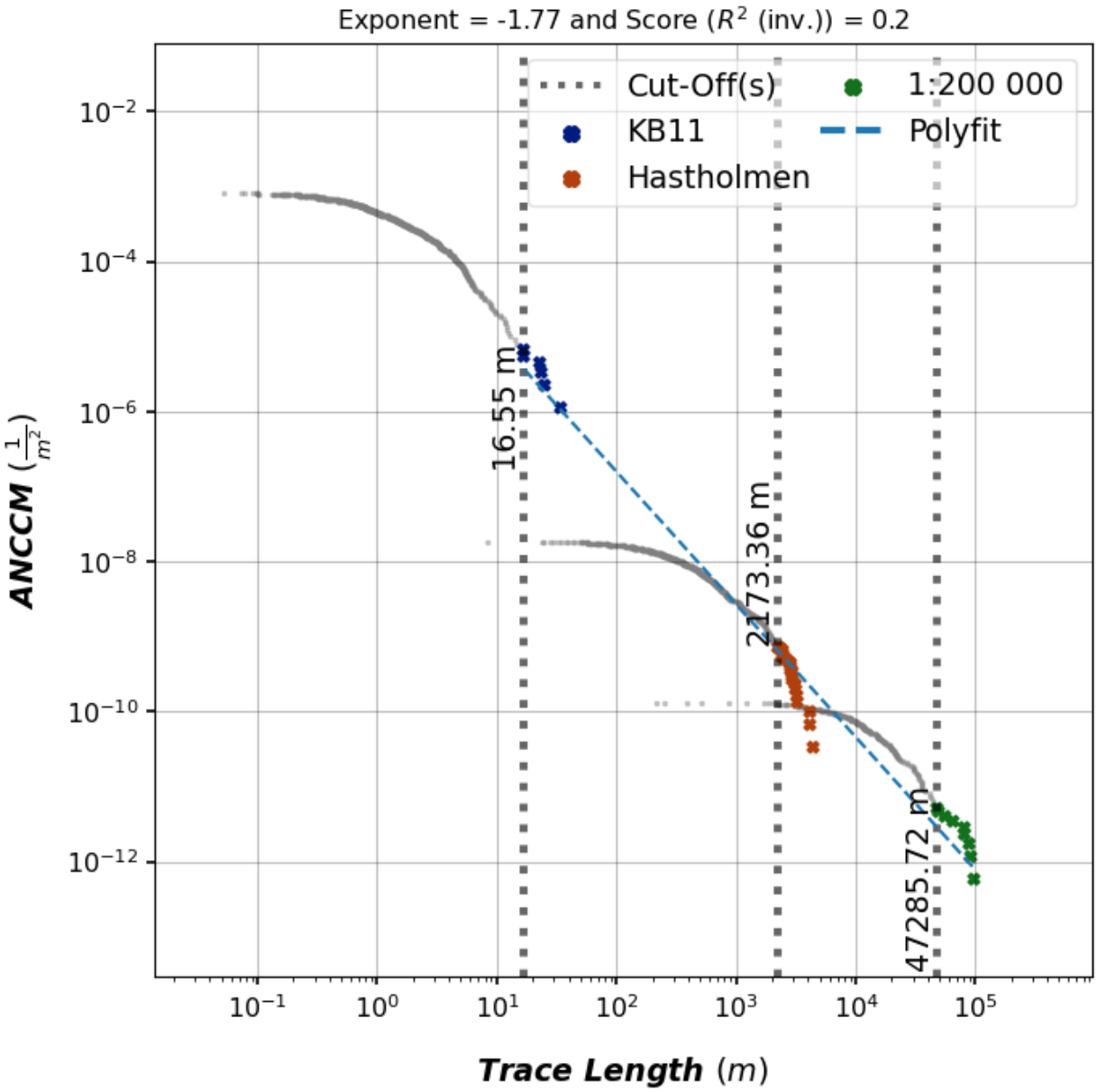
Use scipy.optimize.shgo to optimize cut-off values

```
# See https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.shgo.html
# for potential keyword arguments to pass to the shgo call
# shgo_kwargs are passed as is to ``scipy.optimize.shgo``.
shgo_kwargs = dict(
    sampling_method="sobol",
)

# Choose loss function for optimization. Here r2 is chosen to get a visually
# sensible result but it is generally ill-suited for optimizing cut-offs of
# power-law distributions.
scorer = general.r2_scorer

# Returns new instance of MultiLengthDistribution
# with optimized cut-offs.
```

(continues on next page)

(continued from previous page)

```
opt_result, opt_mld = mld.optimize_cut_offs(scorer=scorer)

# Use optimized MultiLengthDistribution to plot distributions and fit.
# automatic_cut_offs is given as False to use the optimized cut-offs added as
# attributes of the MultiLengthDistribution instance.
polyfit, fig, ax = opt_mld.plot_multi_length_distributions(
    automatic_cut_offs=False, scorer=scorer, plot_truncated_data=True
)

# Print some results
print(
    f"""
Optimized cut-offs:
{opt_result.optimize_result.x}
Resulting power-law exponent:
{opt_result.polyfit.m_value}
Resulting {scorer.__name__} score:
{opt_result.polyfit.score}
"""
)

# Visual plot setup
plt.tight_layout()
```




```
Optimized cut-offs:
[1.65542653e+01 2.17336026e+03 4.72857200e+04]
Resulting power-law exponent:
-1.76977559492765
Resulting r2_scorer score:
0.19959433366828727
```

Total running time of the script: (0 minutes 1.107 seconds)

Module for creating `fractopo.Network` objects for examples

The module loads fracture datasets, e.g. one named KB11, from `fractopo` GitHub repository and creates a `fractopo.Network` object of it. Along with the traces the target area in which the traces have been determined in is required (it is similarly loaded from `fractopo` GitHub page).

Setup matplotlib rcParams for better plotting in the examples

```
# import matplotlib as mpl

# mpl.rcParams["figure.autolayout"] = True
# mpl.rcParams["figure.constrained_layout.use"] = True
# mpl.rcParams["savefig.bbox"] = "tight"
```

Initialize `fractopo.Network`

Note that azimuth sets are explicitly set here for both networks. They are user-defined though the Network will use default azimuth sets if not set by the user.

```
import geopandas as gpd

from fractopo import Network

kb11_network = Network(
    name="KB11",
    trace_gdf=gpd.read_file(
        "https://raw.githubusercontent.com/nialov/"
        "fractopo/master/tests/sample_data/KB11/KB11_traces.geojson"
    ),
    area_gdf=gpd.read_file(
        "https://raw.githubusercontent.com/nialov/"
        "fractopo/master/tests/sample_data/KB11/KB11_area.geojson"
    ),
    truncate_traces=True,
    circular_target_area=False,
    determine_branches_nodes=True,
    snap_threshold=0.001,
    # Explicitly set azimuth sets
    azimuth_set_names=("N-S", "E-W"),
    azimuth_set_ranges=((135, 45), (45, 135)),
)

hastholmen_network = Network(
    name="Hastholmen",
    trace_gdf=gpd.read_file(
        "https://raw.githubusercontent.com/nialov/"
        "fractopo/master/tests/sample_data/hastholmen_traces_validated.geojson"
    ),
    area_gdf=gpd.read_file(
        "https://raw.githubusercontent.com/nialov/"
        "fractopo/master/tests/sample_data/hastholmen_area.geojson"
```

(continues on next page)

(continued from previous page)

```
    ),
    truncate_traces=True,
    circular_target_area=False,
    determine_branches_nodes=True,
    snap_threshold=0.001,
    # Explicitly set azimuth sets (same as for KB11)
    azimuth_set_names=("N-S", "E-W"),
    azimuth_set_ranges=((135, 45), (45, 135)),
)

lidar_200k_network = Network(
    name="1:200 000",
    trace_gdf=gpd.read_file(
        "https://raw.githubusercontent.com/nialov/"
        "fractopo/master/tests/sample_data/traces_200k.geojson"
    ),
    area_gdf=gpd.read_file(
        "https://raw.githubusercontent.com/nialov/"
        "fractopo/master/tests/sample_data/area_200k.geojson"
    ),
    truncate_traces=True,
    circular_target_area=False,
    determine_branches_nodes=True,
    snap_threshold=0.001,
    # Explicitly set azimuth sets (same as for KB11)
    azimuth_set_names=("N-S", "E-W"),
    azimuth_set_ranges=((135, 45), (45, 135)),
)
```

Total running time of the script: (0 minutes 0.000 seconds)

Visualize different types of validation errors

This example demonstrates using `shapely` and `geopandas` for visualization purposes. The different types of trace validation error scenarios are created declaratively in code with `shapely` geometries (`LineStrings`). Just scroll down to the bottom if you are interested in the visualization output.

Imports

```
import logging
from pathlib import Path
from typing import Sequence

import geopandas as gpd
import matplotlib.pyplot as plt
from matplotlib.axes._axes import Axes
from matplotlib.figure import Figure

# Import the geometries used to create traces and target areas.
from shapely.geometry import LineString, Point
```

Traces are explicitly defined in code and plotted

```
# Initialization
fig: Figure
fig, axes = plt.subplots(2, 3, figsize=(7, 7))
fig.tight_layout(h_pad=1.5)

axes_flat: Sequence[Axes] = axes.flatten()


for ax in axes_flat:
    ax.set_xlim(-6, 6)
    ax.set_ylim(-6, 6)
    ax.axis("off")
    # ax.text(x=-5, y=5, s=next(labeler), fontsize="x-large", fontweight="bold")

default_text_kwargs = dict(ha="center", fontdict=dict(size="small"))

# Axis 1: MULTI JUNCTION

axis_1 = axes_flat[0]

traces_1 = gpd.GeoDataFrame(
    geometry=[
        LineString([(-5, 0), (5, 0)]),
        LineString([(0, 5), (0, -5)]),
        LineString([(-2.5, 2.5), (2.5, -2.5)]),
    ]
)
errors_1 = gpd.GeoDataFrame(geometry=[Point(0, 0)])
traces_1.plot(ax=axis_1, color="black")
errors_1.plot(ax=axis_1, marker="X", color="red", zorder=10)
axis_1.text(
    x=0,
    y=-7,
    s="More than two traces intersect\non the same point.",
    **default_text_kwargs,
)

# Axis 2: MULTI JUNCTION

axis_2 = axes_flat[1]

traces_2 = gpd.GeoDataFrame(
    geometry=[
        LineString([(-2.5, 2.5), (0.5, -0.5)]),
        LineString([(-5, -5), (5, 5)]),
    ]
)
traces_2.plot(ax=axis_2, color="black")
axis_2.annotate(
    "Overlap distance higher\n than defined snap threshold.",
```

(continues on next page)

(continued from previous page)

```
    xy=(0.5, -0.5),
    xytext=(-1.0, -4),
    arrowprops=dict(arrowstyle="->", color="red"),
    fontstyle="italic",
    fontsize="small",
)
# axis_2.set_title("Trace overlaps another trace.")
axis_2.text(x=0, y=-7, s="Trace overlaps another trace.", **default_text_kwargs)

# Axis 3: V NODE

axis_3 = axes_flat[2]

traces_3 = gpd.GeoDataFrame(
    geometry=[
        LineString([(-2.5, 2.5), (0, 0)]),
        LineString([(-2.5, -5), (0, 0)]),
    ]
)
errors_3 = gpd.GeoDataFrame(geometry=[Point(0, 0)])
traces_3.plot(ax=axis_3, color="black")
errors_3.plot(ax=axis_3, marker="X", color="red", zorder=10)
# axis_3.set_title("Two traces end in a\nV-node formation.")
axis_3.text(
    x=0,
    y=-7,
    s="Two traces end in a\nV-node formation.",
    **default_text_kwargs,
)

# Axis 4: MULTIPLE CROSSCUTS

axis_4 = axes_flat[3]

traces_4 = gpd.GeoDataFrame(
    geometry=[
        LineString([(-5, 0), (5, 0)]),
        LineString([(-5, 1), (-2, -1), (1, 1), (5, -1)]),
    ]
)
intersections = traces_4.geometry.values[0].intersection(traces_4.geometry.values[1])

errors_4 = gpd.GeoDataFrame(geometry=list(intersections.geoms))

traces_4.plot(ax=axis_4, color="black")
errors_4.plot(ax=axis_4, marker="X", color="red", zorder=10)
# axis_4.set_title("Two traces cross each\nother more than two times.")
axis_4.text(
    x=0,
    y=-7,
    s="Two traces cross each\nother more than two times.",
    **default_text_kwargs,
```

(continues on next page)

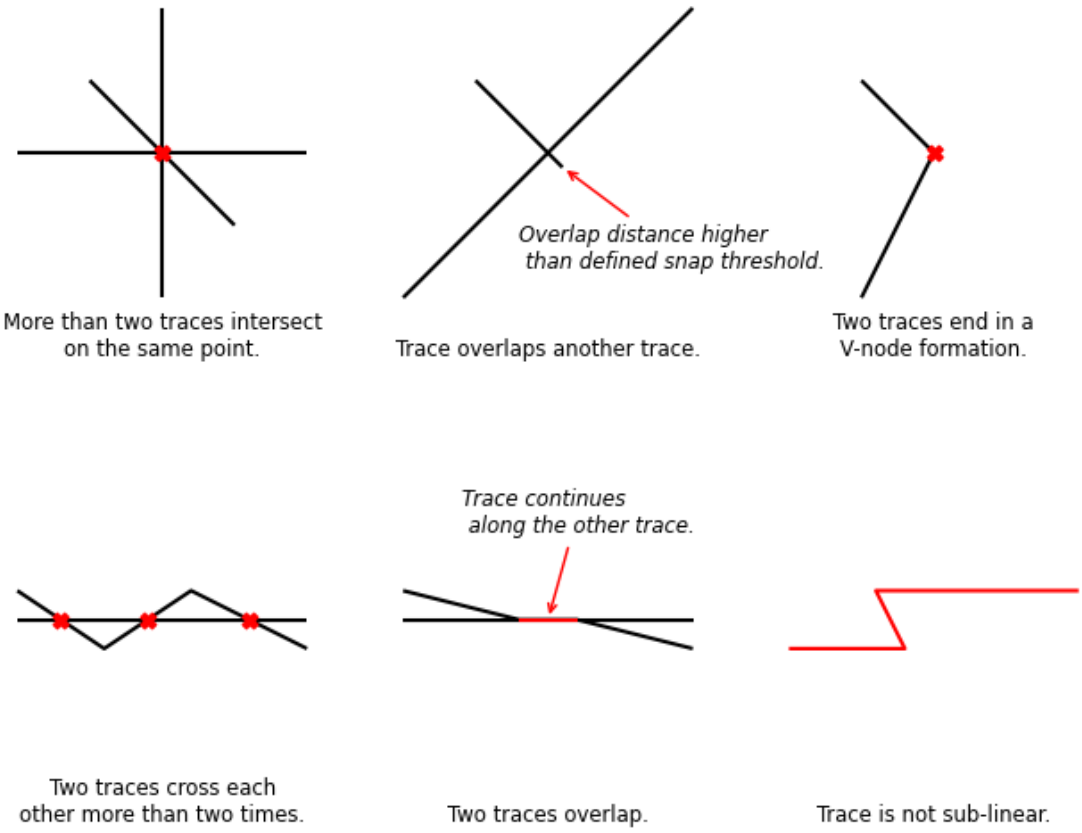
(continued from previous page)

```
)

# Axis 5: OVERLAPPING

axis_5 = axes_flat[4]

traces_5 = gpd.GeoDataFrame(
    geometry=[
        LineString([(-5, 0), (5, 0)]),
        LineString([(-5, 1), (-1, 0), (1, 0), (5, -1)]),
    ]
)
intersections = traces_5.geometry.values[0].intersection(traces_5.geometry.values[1])
assert isinstance(intersections, LineString)

errors_5 = gpd.GeoDataFrame(geometry=[intersections])

traces_5.plot(ax=axis_5, color="black")
errors_5.plot(ax=axis_5, color="red", zorder=10)
# axis_5.set_title("Two traces cross each\nother more than two times.")
axis_5.text(
    x=0,
    y=-7,
    s="Two traces overlap.",
    **default_text_kwargs,
)
axis_5.annotate(
    "Trace continues\n along the other trace.",
    xy=(0.0, 0.0),
    xytext=(-3, 3),
    arrowprops=dict(arrowstyle="->", color="red"),
    fontstyle="italic",
    fontsize="small",
)

# Axis 6: OVERLAPPING

axis_6 = axes_flat[5]

traces_6 = gpd.GeoDataFrame(
    geometry=[
        LineString([(-5, -1), (-1, -1), (-2, 1), (5, 1)]),
    ]
)

traces_6.plot(ax=axis_6, color="red")
# axis_6.set_title("Two traces cross each\nother more than two times.")
axis_6.text(
    x=0,
    y=-7,
    s="Trace is not sub-linear.",
    **default_text_kwargs,
```

(continues on next page)

(continued from previous page)

```
)
# axis_6.annotate(
#     "Trace continues\n along the other trace.",
#     xy=(0.0, 0.0),
#     xytext=(-3, 3),
#     arrowprops=dict(arrowstyle="->", color="red"),
#     fontstyle="italic",
#     fontsize="small",
# )

plt.subplots_adjust(wspace=0.11, hspace=-0.31)

if __name__ == "__main__":
    # Save plot for usage outside sphinx
    # This section can be ignored if looking at the documentation
    # in ReadTheDocs
    output_name = "validation_errors.png"
    try:
        output_path = Path(__file__).parent / output_name
        fig.savefig(output_path, bbox_inches="tight")
    except Exception:
        # Log error due to e.g. execution as jupyter notebook
        logging.info(f"Failed to save {output_name} plot.", exc_info=True)
```



Total running time of the script: (0 minutes 0.568 seconds)

Script to create a workflow visualization of `fractopo`.

```
import sys
from pathlib import Path
from string import ascii_uppercase
from tempfile import TemporaryDirectory
from typing import Callable, Optional, Tuple

import matplotlib.pyplot as plt
from example_networks import kb11_network
from matplotlib.axes import Axes
from matplotlib.figure import Figure
```

(continues on next page)

(continued from previous page)

```
from matplotlib.patheffects import withStroke
from PIL import Image

from fractopo.analysis.network import Network, assign_branch_and_node_colors


def close_fig(func: Callable):
    """
    Wrap function to close any ``matplotlib`` plots after call.
    """

    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        plt.close()
        return result

    return wrapper


def plot_area(kb11_network: Network = kb11_network) -> Figure:
    """
    Plot area boundary.
    """
    area_fig, area_ax = plt.subplots()
    # Target area
    kb11_network.area_gdf.boundary.plot(ax=area_ax, color="red")
    # Annotate A to B
    area_ax.axis("off")
    return area_fig


def plot_traces_and_area(kb11_network: Network = kb11_network) -> Figure:
    """
    Plot area boundary along with the traces.
    """
    area_fig, traces_and_area_ax = plt.subplots()
    # Target area

    # Traces and area
    kb11_network.area_gdf.boundary.plot(ax=traces_and_area_ax, color="red")
    kb11_network.trace_gdf.plot(ax=traces_and_area_ax, color="blue")
    traces_and_area_ax.axis("off")
    return area_fig


def plot_branches_and_area(kb11_network: Network = kb11_network) -> Figure:
    """
    Plot area boundary along with the branches.
    """
    area_fig, branches_and_area_ax = plt.subplots()
    # Target area
```

(continues on next page)

(continued from previous page)

```
    # Traces and area
    kb11_network.area_gdf.boundary.plot(ax=branches_and_area_ax, color="red")
    kb11_network.branch_gdf.plot(
        colors=[assign_branch_and_node_colors(bt) for bt in kb11_network.branch_types],
        ax=branches_and_area_ax,
    )
    branches_and_area_ax.axis("off")
    return area_fig


def plot_rose() -> Figure:
    """
    Plot rose plot.
    """
    import plot_rose_plot

    return plot_rose_plot.fig


def plot_ortho(ax: Axes):
    """
    Plot drone orthomosaic.
    """
    # Ortho
    image_path = Path(__file__).parent.parent / "docs_src/imgs/kb11_orthomosaic.jpg"
    with Image.open(image_path) as image:
        # image = image.convert("RGBA")
        # data = np.ones((image.size[1], image.size[0], 4), dtype=np.uint8) * 255
        # mask = Image.fromarray(data)
        # mask.paste(image, (0, 0))
        ax.imshow(image)


def plot_lengths():
    """
    Plot length distribution plot.
    """
    import plot_length_distributions

    return plot_length_distributions.fig


def plot_xyi():
    """
    Plot XYI node count ternary plot.
    """
    import plot_ternary_plots

    return plot_ternary_plots.xyi_fig


def plot_branches():
```

(continues on next page)

(continued from previous page)

```
    """
    Plot branch count ternary plot.
    """
    import plot_ternary_plots

    return plot_ternary_plots.branch_fig


def plot_relationships():
    """
    Plot azimuth set relationships plot.
    """
    import plot_azimuth_set_relationships

    return plot_azimuth_set_relationships.figs[0]


def add_plot_image_to_ax(figure: Figure, ax: Axes):
    """
    Add a disk-saved plot to an ax.
    """
    with TemporaryDirectory() as tmp_dir:
        save_path = Path(tmp_dir) / "plot.png"
        figure.savefig(save_path, bbox_inches="tight")
        with Image.open(save_path) as image:
            ax.imshow(image)


def arrow_annotation(ax: Axes, start: Tuple[float, float], end: Tuple[float, float]):
    """
    Annotate ax with arrow.
    """
    ax.annotate(
        "",
        xy=end,
        xycoords="axes fraction",
        xytext=start,
        textcoords="axes fraction",
        arrowprops=dict(facecolor="black", shrink=2.05, width=0.1),
        horizontalalignment="center",
        verticalalignment="top",
        zorder=100,
    )


def main(output_path: Optional[Path] = None):
    # Initialize figure with 3x3 grid
    figure, axes = plt.subplots(3, 3, figsize=(9, 9))
    assert isinstance(figure, Figure)
    axes_flatten = axes.flatten(order="F")

    # Give explicit names to axes
```

(continues on next page)

(continued from previous page)

```
    (
        area_ax,
        traces_and_area_ax,
        ortho_ax,
        rose_ax,
        length_ax,
        branches_and_area_ax,
        relationships_ax,
        xyi_ax,
        branch_ax,
    ) = axes_flatten

    # Compose the image by adding disk-written images of plots with imshow to all the␣
↪axes
    add_plot_image_to_ax(ax=area_ax, figure=plot_area())
    add_plot_image_to_ax(ax=traces_and_area_ax, figure=plot_traces_and_area())
    # Orthomosaic is already an image on the disk
    plot_ortho(ax=ortho_ax)
    add_plot_image_to_ax(ax=rose_ax, figure=plot_rose())
    add_plot_image_to_ax(ax=length_ax, figure=plot_lengths())
    add_plot_image_to_ax(ax=relationships_ax, figure=plot_relationships())
    add_plot_image_to_ax(ax=branches_and_area_ax, figure=plot_branches_and_area())
    add_plot_image_to_ax(ax=branch_ax, figure=plot_branches())
    add_plot_image_to_ax(ax=xyi_ax, figure=plot_xyi())

    # Final annotations and setup

    ## Arrows
    arrow_annotation(ax=ortho_ax, start=(0.5, 0.9), end=(0.5, 1.2))
    arrow_annotation(ax=area_ax, start=(0.5, 0.1), end=(0.5, -0.2))

    # Traces and area to data
    arrow_annotation(ax=traces_and_area_ax, start=(0.95, 0.5), end=(1.2, 1.5))
    arrow_annotation(ax=traces_and_area_ax, start=(0.95, 0.5), end=(1.2, 0.5))
    arrow_annotation(ax=traces_and_area_ax, start=(0.95, 0.5), end=(1.2, -0.5))

    # Branches to xyi and branch ternary
    arrow_annotation(ax=branches_and_area_ax, start=(0.9, 0.5), end=(1.2, 0.5))
    arrow_annotation(ax=branches_and_area_ax, start=(0.9, 0.5), end=(1.2, 1.35))

    # Rose to relationships
    arrow_annotation(ax=rose_ax, start=(0.9, 0.5), end=(1.1, 0.5))

    area_ax.text(
        x=0.37,
        y=-0.04,
        s="Define target area",
        rotation=90,
        transform=area_ax.transAxes,
        va="center",
    )
    ortho_ax.text(
```

(continues on next page)

(continued from previous page)

```
        x=0.55,
        y=1.1,
        s="Digitize traces",
        rotation=90,
        transform=ortho_ax.transAxes,
        va="center",
    )
    traces_and_area_ax.text(
        x=1.05,
        y=1.1,
        s="Orientation/Azimuth",
        rotation=75,
        transform=traces_and_area_ax.transAxes,
        va="center",
        ha="center",
    )
    traces_and_area_ax.text(
        x=1.12,
        y=0.05,
        s="Topology",
        rotation=-75,
        transform=traces_and_area_ax.transAxes,
        va="center",
        ha="center",
    )
    branches_and_area_ax.text(
        x=1.07,
        y=0.4,
        s="Branches",
        # rotation=90,
        transform=branches_and_area_ax.transAxes,
        va="center",
        ha="center",
    )
    branches_and_area_ax.text(
        x=1.0,
        y=1.0,
        s="Nodes",
        rotation=70,
        transform=branches_and_area_ax.transAxes,
        va="center",
        ha="center",
    )

    ## Labels
    for idx, ax in enumerate(axes_flatten):
        assert isinstance(ax, Axes)
        text = ax.text(
            x=0.1,
            y=0.95,
            s=f"{ascii_uppercase[idx]}.",
            transform=ax.transAxes,
```

(continues on next page)

(continued from previous page)

```
            fontsize="xx-large",
        )
        ax.axis("equal")
        text.set_path_effects(
            [
                withStroke(linewidth=2, foreground="white"),
            ]
        )
        ax.axis("off")

    plt.close("all")
    if output_path is not None:
        print(f"Saving workflow plot to {output_path}")
        figure.savefig(output_path, bbox_inches="tight", dpi=150)


if __name__ == "__main__":
    if len(sys.argv) > 1:
        output_path = Path(sys.argv[1])
    else:
        output_path = None
    main(output_path=output_path)
```

Total running time of the script: (0 minutes 0.000 seconds)

6.1.5 Validating fracture trace data

Prerequisites

Fracture or lineament trace data is typically digitized in a GIS environment. The basemap on which fractures are digitized could be an orthomosaic or a Light Detection and Ranging digital elevation model for lineaments. If topology of the fracture network is to be analyzed the constraints it poses on digitization must be taken into account. E.g. traces must be accurately snapped to end to other traces to form a Y-node (See Sanderson and Nixon, 2015).

Alongside trace data a defined target/sample area (or areas) must be supplied.

All spatial file types containing two-dimensional polylines (i.e. `LineStrings`) that can be loaded with geopandas can be validated as trace data. The validation tool along with all other fractopo modules only accept `geopandas.GeoDataFrame's` as inputs (`geopandas` easily handles transformation of spatial data types – shapefiles, geopackages, etc. – to `GeoDataFrames` and back). A `GeoDataFrame` is the inmemory representation of the spatial data that is modified in a Python session.

Validation

Validation consists of finding errors in digitization and then fixing them. However currently only very few error types are automatically fixed. Instead it is recommended to use the validation tool to find the errors and then fixing them manually. The tool creates a new column, *VALIDATION ERRORS*, in the `GeoDataFrame` (visible in the attribute table in GIS-software). Currently very few types of errors can be automatically fixed and e.g. conversion from `LineString` to `MultiLineString` has to be done to allow further validation. Therefore I currently recommend allowing automatic fixes when prompted.

Page links below explain how to use the validation tool in Python, the validation error types and how manually fix the validation errors.

- Usage examples in two notebooks:
  - Trace data validation 1
  - Trace data validation 2
  - Loading data for analysis with `geopandas`
  - Trace validation
  - Visualizing found validation errors with `geopandas` and `matplotlib`

For validation error descriptions:

- Validation Error Types

6.1.6 Validation errors

The error string is put into the error column of the validated trace data. Each error string is explained below. Possible automatic fixing is indicated by the checkmarks:

No automatic fixing:

```
* [ ] Automatic fix
```

Some cases can be automatically fixed:

```
* [o] Automatic fix
```

All cases can be automatically fixed:

```
* [X] Automatic fix
```

This page additionally serves as a reminder on what interpretations to avoid while digitizing traces. Most of the validation errors displayed here cause issues in further analyses and should be fixed before attempting to e.g. determine branches and nodes.

Only exception to the normal validation procedure is the check for empty target areas (when target area is passed without any traces intersecting it) and `allow_empty_area` is set to `False` (defaults to `True` i.e. no check).

In this case the error string is in all traces.

The empty area error string is:

```
"EMPTY TARGET AREA"
```

GeomTypeValidator

The error string is:

```
"GEOM TYPE MULTILINESTRING"
```

The error is caused by the geometry type which is wrong: `MultiLineString` instead of `LineString`.

`MultiLineString` can consist of multiple `LineStrings` i.e. a `MultiLineString` can consist of disjointed traces. A `LineString` only consists of a single continuous trace.

- [o] Automatic fix:
  - Mergeable `MultiLineStrings`

  - `MultiLineStrings` with a single `LineString`

Most of the time `MultiLineStrings` are created instead of `LineStrings` by the GIS-software and the `MultiLineStrings` actually only consist of a single `LineString` and conversion from a `MultiLineString` with a single `LineString` to a `LineString` can be done automatically. If the `MultiLineString` does consist of multiple `LineStrings` they can be automatically merged if they are not disjointed i.e. the contained `LineStrings` join together into a single `LineString`. If they cannot be automatically merged no automatic fix is performed and the error is kept in the error column and the user should fix the issue.

`MultiLineStrings` are not accepted by `fractopo` as they cause 1. a physical and 2. a technical inconsistency.

1. A single fracture trace in `fractopo` is considered to be continuous and sublinear. A `LineString` fulfils this criteria better than a `MultiLineString` as they are always continuous.
2. If `MultiLineStrings` and `LineStrings` are mixed in data, then it becomes inconsistent in if geometries are one or many in the rows of a `geopandas.GeoDataFrame`.

MultiJunctionValidator

The error string is:

```
"MULTI JUNCTION"
```

Three error types can occur in digitization resulting in this error string:

1. More than two traces must not cross in the same point or **too close** to the same point.
2. An overlapping Y-node i.e. a trace overlaps the trace it "is supposed" to end at too much (alternatively detected by *UnderlappingSnapValidator*).
3. *V NODE* errors might also be detected as `MULTI JUNCTION` errors.

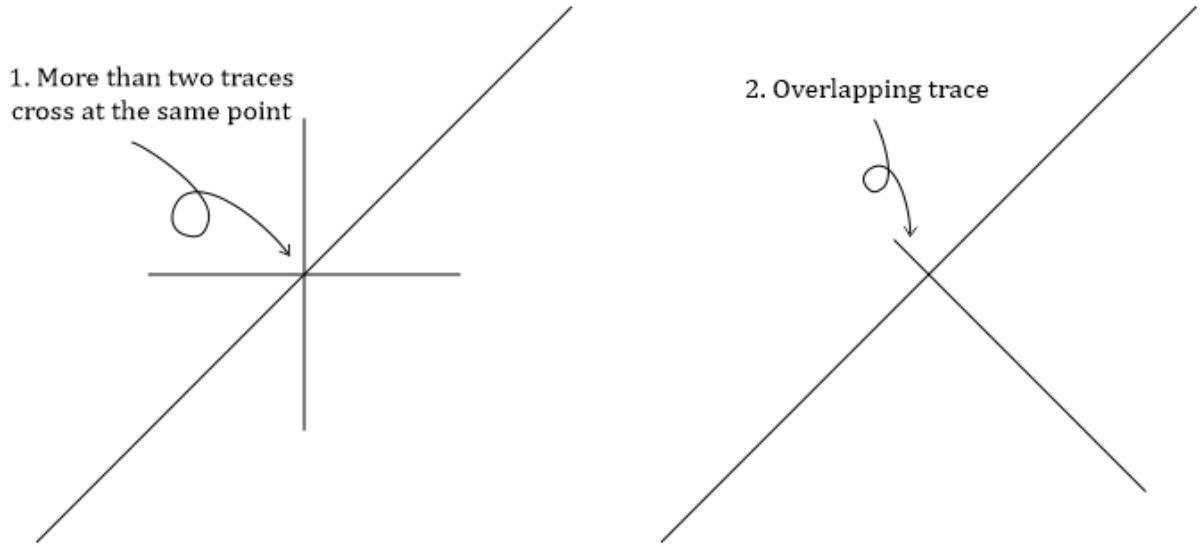


Fig. 1: Multi junction error examples.

- [ ] Automatic fix

Fix the error manually by making sure neither of the above rules are broken.

Motivation for this validation is primarily physical. An intersection between three or more fractures is not defined as a separate topological node as it is practically impossible for three-dimensional fracture planes to have a common point. Consequently, there should never be an intersection point between three fracture traces in nature and such points are highlighted by this validation error. For the overlapping and V-node errors see the respective error classes for motivation.

VNodeValidator

The error string is:

```
"V NODE"
```

Two traces end at the same point or close enough to be interpreted as the same endpoint.

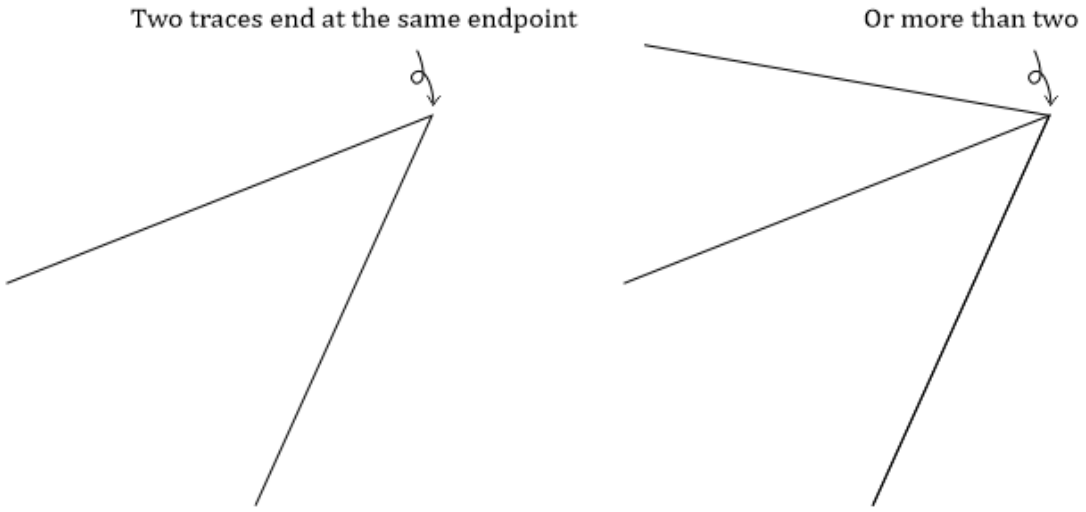


Fig. 2: V-node error examples.

- [ ] Automatic fix

Fix by making sure two traces never end too near to each other.

V-nodes are not physically possible as it would require two fracture planes to end along a single line and are not defined as a topological node by Sanderson and Nixon (2015).

MultipleCrosscutValidator

The error string is:

```
"MULTIPLE CROSSCUTS"
```

Two traces cross each other more than two times i.e. they have geometrically more than two common coordinate points.

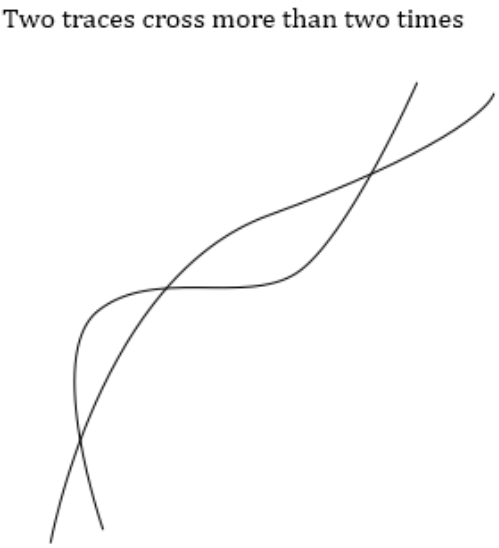


Fig. 3: Multiple crosscut error examples.

- [ ] Automatic fix

Fix by decreasing the number of crosses to a maximum of two between two traces.

Crosscuts between two traces more than two times more than likely indicate either digitizing errors or e.g. topographical errors within a raster where the traces have been digitized from. Due to topography, the fracture plane intersections with the surface might appear curved even though the plane itself might be subplanar.

However, it is probably possible for curved fracture surfaces to crosscut each other more than two times in nature. However, this is considered sufficiently rare that `fractopo` will not accept these as valid fracture trace data. If such data needs to be analysed in the future, the functionality can be considered to be added (post an issue to `fractopo` GitHub!) and this validation error might be reworked.

UnderlappingSnapValidator

The error string is:

```
"UNDERLAPPING SNAP"
```

Or:

```
"OVERLAPPING SNAP"
```

Underlapping error can occur when a trace ends very close to another trace but not near enough. The abutting might not be registered as a Y-node.

Overlapping error can occur when a trace overlaps another only very slightly resulting in a dangling end. Such dangling ends might not be registered as Y-nodes and might cause spatial/topological analysis problems later.

Overlapping snap might also be registered as a *MULTI JUNCTION* error.

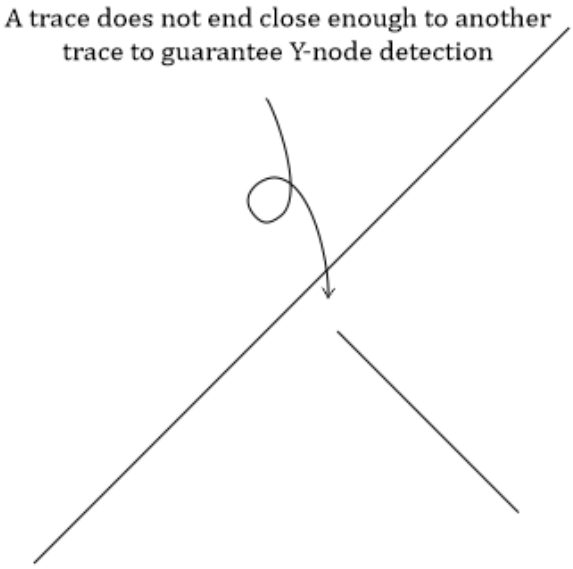


Fig. 4: Underlapping snap error examples.

- [ ] Automatic fix

Fix by more accurately snapping the trace to the other trace.

The motivation for this validation error comes from two technical problems:

1. Even though a trace abuts outside the snap threshold, and the abutment is not registered as a Y-node, the proximity can still cause errors in analysis during e.g. crosscutting and abutting relationship detection.
2. It is very unlikely, that abutments that occur slightly outside the snap threshold (the detected errors) are intentional. Rather, they often highlight errors during digitizing where an abutment has failed in the GIS-software used for digitizing.

Consequently, `UnderlappingSnapValidator` provides a safety buffer for more accurate analysis of topological relationships.

TargetAreaSnapValidator

The error string is:

```
"TRACE UNDERLAPS TARGET AREA"
```

A trace ends very close to the edge of the target area but not close enough. The abutting might not be registered as a E-node i.e. a trace endpoint that ends at the target area. E-nodes indicate that the trace length is undetermined.

- [ ] Automatic fix

Fix by extending the trace over the target area. The analyses typically crop the traces to the target area so there's very little reason not to always extend over the target area edge.

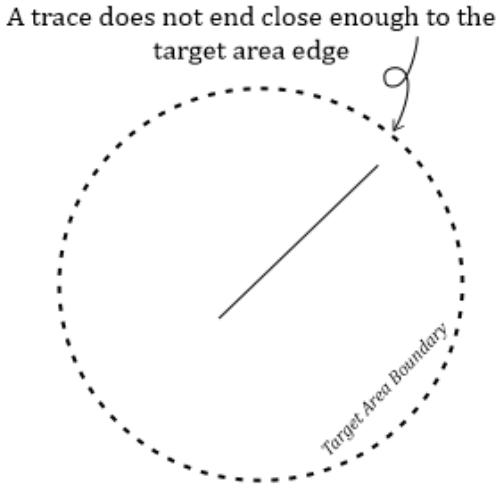


Fig. 5: Target area snap error examples.

This technical error is expected to occur during faulty digitizing and fixing these errors helps avoid inconsistent analysis regarding E-nodes.

GeomNullValidator

The error string is:

```
"NULL GEOMETRY"
```

Rows with geometry set to None or equivalent type that is not a valid GIS geometry or rows with empty geometries.

These rows could be automatically removed but these are most likely rare occurrences and deleting the row would cause all attribute data associated with the row to be consequently removed.

- [ ] Automatic fix

Fix by deleting the row or creating a geometry for the row. GIS software can be fickle with these, make sure that if you create a new geometry it gets associated to the row in question.

StackedTracesValidator

The error string is:

```
"STACKED TRACES"
```

Two (or more) traces are stacked partially or completely on top of each other. Also finds cases in which two traces form a very small triangle intersection.

- [ ] Automatic fix

Fix by editing traces do that they do not stack or intersect in a way to create small triangles.

Stacked traces are the result of faulty digitizing as two fractures cannot co-exist along the same trace or that is at least the technical and physical expectation in `fractopo`.

SimpleGeometryValidator

The error string is:

```
"CUTS ITSELF"
```

A trace intersects itself.

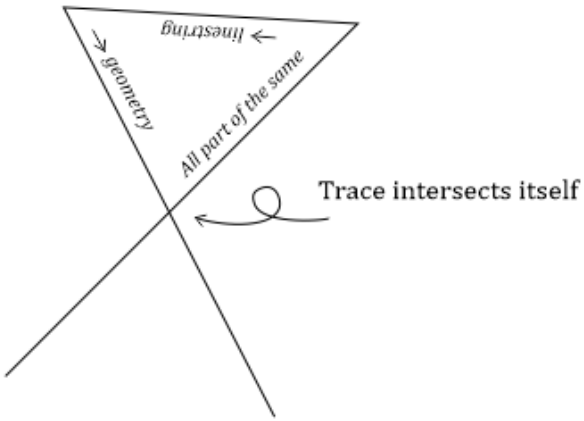


Fig. 6: Trace intersects itself.

- [ ] Automatic fix

Fix by removing self-intersections.

Fracture planes in nature should be subplanar and therefore unable to cut themselves.

SharpCornerValidator

The error string is:

```
"SHARP TURNS"
```

A lineament or fracture trace should not make erratic turns and the trace should be sublinear. The exact limit on of what is erratic and what is not is **open to interpretation and therefore the resulting errors are subjective**. But if a segment of a trace has a direction change of over 180 degrees compared to the previous there is probably no natural way for a natural bedrock structure to do that.

`SHARP TURNS` -errors rarely cause issues in further analyses. Therefore fixing these issues is not critical.

- [ ] Automatic fix

Fix (if desired) by making less sharp turns and making sure the trace is sublinear.

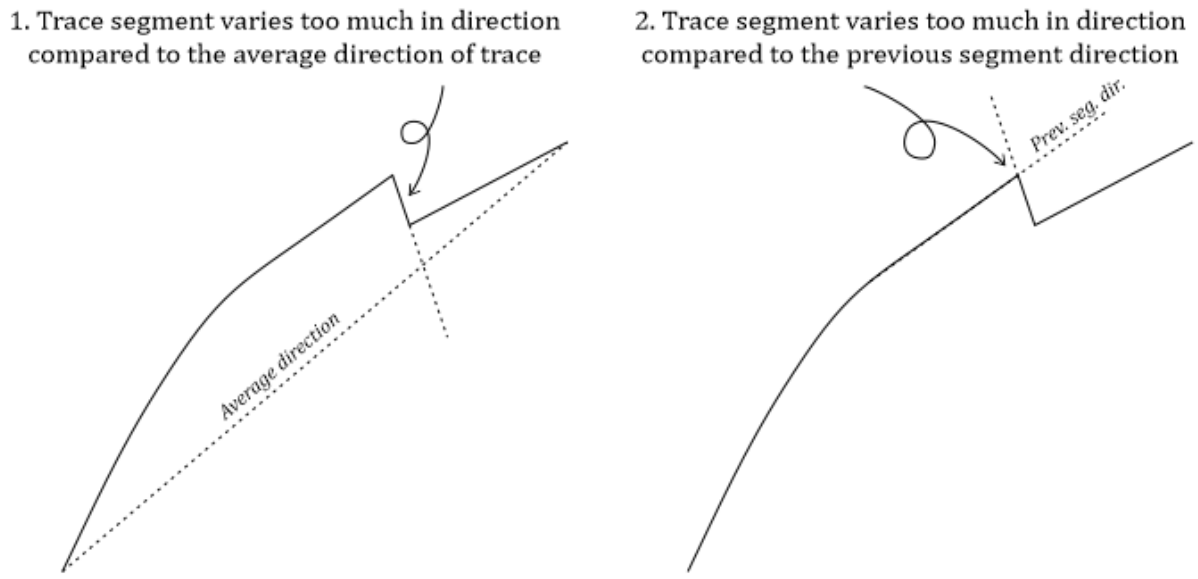


Fig. 7: Erratic trace segment direction change examples.

6.1.7 Miscellanous/FAQ

Geometries with Z-coordinates

Currently `fractopo` has only partly been tested with geometries that have Z-coordinates (e.g. `LineString([(0, 0, 0), (1, 1, 1)])`. Any issues you believe might be related to Z-coordinates should be reported on *GitHub*. To test for such issues using your data, try removing the Z-coordinates with e.g. the following code:

```
from shapely.geometry import Polygon, MultiPolygon, LineString


def remove_z_coordinates(geometries):
    """
    :param geometries: Shapely geometry iterable
    :return: a list of Shapely geometries without z coordinates
    """
    return [
        shapely.wkb.loads(shapely.wkb.dumps(geometry, output_dimension=2))
        for geometry in geometries
    ]


trace_data.geometry = remove_z_coordinates(trace_data.geometry)
```

Thanks to *Siyh* for the issue report and above code example (https://github.com/nialov/fractopo/issues/21).

Furthermore, some validation and analysis remove Z-coordinates from geometric results. Specifically, use of `Validation` will remove Z-coordinates to avoid unexpected behaviour. Use of `Network` will by default remove z-coordinates but this can be disabled (See `Network` input arguments)..

Snap threshold parameter

Both in validation and analysis a `snap_threshold` parameter is used. The `snap_threshold` is for the most part designed to handle the (unavoidable) errors in topological snapping. If the traces in the map are snapped using the snapping functionality of e.g. QGIS or ArcGIS, the distance between endpoints and the segments they should be snapped to be should be well below the threshold of 0.001, which is the default value used in `fractopo`. This threshold during digitising can probably be changed in the settings of these software but for the most part the snapping error should not be connected to e.g. the lengths of the traces you are digitising.

I would recommend, rather than using a higher `snap_threshold` in `fractopo`, to make sure that the snapping is done properly within the GIS software used in digitising.

Coordinate systems

`fractopo` for the most part assumes a metric coordinate system and has mostly been tested with data which has the coordinate system of EPSG 3057. Issues with data from other coordinate systems should be posted on *GitHub*. Using a coordinate system with metric units should be the first step in debugging if you believe an issue is caused by `fractopo` not handling other units correctly. Especially the `snap_threshold` value should be adjusted based on the units used.

6.1.8 Contributing Guidelines

Contributions to `fractopo` are welcome and highly appreciated. Contributions do not have to be pull requests (i.e., code contributions), rather it is very useful for you to report any issues and problems you have in installing, using the software and misunderstandings that might arise due to lacking or misleading documentation.

If you are up for submitting code, an issue is first recommended to be created about your plans for the submission so it can be determined if the pull request is deemed suitable for the project and subsequently, time will not be wasted making pull requests that are not suitable.

In particular, when submitting a pull request:

- Install the requirements for the project using `poetry`. `poetry` uses the `pyproject.toml` and `poetry.lock` files to replicate the development environment.
- Run the unit tests using `pytest` i.e., `poetry run pytest`. Some tests are very strict (regression tests i.e. the result should match exactly the previous test results) and therefore no strict requirements are made that all tests should always pass for every pull request. A few failing tests can always be reviewed in the pull request phase!
- Please write new tests if functionality is added or changed!

Style

- See `pyproject.toml` for the supported Python versions
- You should set up pre-commit hooks to automatically run a number of style and syntax checks on the code. The `pre-commit` checks are the primary validation for the style of the code.
- New code should be documented following the style of the code base i.e., using the `sphinx` style.

6.1.9 fractopo package

Subpackages

fractopo.analysis package

Submodules

fractopo.analysis.anisotropy module

Anisotropy of connectivity determination utilities.

fractopo.analysis.anisotropy.**determine_anisotropy_classification**(*branch_classification*)

Return value based on branch classification.

Only C-C branches have a value, but this can be changed here. Classification can differ from 'C - C', 'C - I', 'I - I' (e.g. 'C - E') in which case a value (0) is still returned.

Parameters
branch_classification (str) – Branch classification string.

Return type
int

Returns
Classification encoded as integer.

E.g.

```
>>> determine_anisotropy_classification("C - C")
1
```

```
>>> determine_anisotropy_classification("C - E")
0
```

fractopo.analysis.anisotropy.**determine_anisotropy_sum**(*azimuth_array*, *branch_types*, *length_array*, *sample_intervals=array([0, 30, 60, 90, 120, 150])*)

Determine the sums of branch anisotropies.

Parameters

- **azimuth_array** (ndarray) – Array of branch azimuth values.
- **branch_types** (ndarray) – Array of branch type classication strings.
- **length_array** (ndarray) – Array of branch lengths.
- **sample_intervals** (ndarray) – Array of the sampling intervals.

Return type
Tuple[ndarray, ndarray]

Returns
Sums of branch anisotropies.

E.g.

```
>>> from pprint import pprint
>>> azimuths = np.array([20, 50, 60, 70])
>>> lengths = np.array([2, 5, 6, 7])
>>> branch_types = np.array(["C - C", "C - C", "C - C", "C - I"])
>>> pprint(determine_anisotropy_sum(azimuths, branch_types, lengths))
(array([ 8.09332329, 11.86423103, 12.45612765,  9.71041492,  5.05739707,
        2.15381611]),
 array([  0,  30,  60,  90, 120, 150]))
```

fractopo.analysis.anisotropy.**determine_anisotropy_value**(*azimuth*, *branch_type*, *length*, *sample_intervals=array([0, 30, 60, 90, 120, 150])*)

Calculate anisotropy of connectivity for a branch.

Based on azimuth, branch_type and length. Value is calculated for preset angles (sample_intervals = np.arange(0, 179, 30))

E.g.

Anisotropy for a C-C classified branch:

```
>>> determine_anisotropy_value(50, "C - C", 1)
:rtype: :py:class:`~numpy.ndarray`
```

array([0.64278761, 0.93969262, 0.98480775, 0.76604444, 0.34202014,
0.17364818])

Other classification for branch:

```
>>> determine_anisotropy_value(50, "C - I", 1)
array([0, 0, 0, 0, 0, 0])
```

fractopo.analysis.anisotropy.**plot_anisotropy_ax**(*anisotropy_sum*, *ax*, *sample_intervals=array([0, 30, 60, 90, 120, 150])*)

Plot a styled anisotropy of connectivity to a given PolarAxes.

Spline done with: https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.CubicSpline.html

fractopo.analysis.anisotropy.**plot_anisotropy_plot**(*anisotropy_sum*, *sample_intervals*)

Plot anisotropy values to new figure.

Return type
Tuple[Figure, Axes]

fractopo.analysis.azimuth module

Functions for plotting rose plots.

class fractopo.analysis.azimuth.**AzimuthBins**(*bin_width*, *bin_locs*, *bin_heights*)

Bases: object

Dataclass for azimuth rose plot bin data.

bin_heights: ndarray

bin_locs: ndarray

bin_width: float

fractopo.analysis.azimuth.**decorate_azimuth_ax**(*ax*, *label*, *length_array*, *set_array*, *set_names*, *set_ranges*, *axial*, *visualize_sets*, *append_azimuth_set_text=False*, *add_abundance_order=False*)

Decorate azimuth rose plot ax.

fractopo.analysis.azimuth.**determine_azimuth_bins**(*azimuth_array*, *length_array=None*, *bin_multiplier=1*, *axial=True*)

Calculate azimuth bins for plotting azimuth rose plots.

E.g.

```
>>> azimuth_array = np.array([25, 50, 145, 160])
>>> length_array = np.array([5, 5, 10, 60])
>>> azi_bins = determine_azimuth_bins(azimuth_array, length_array)
>>> azi_bins.bin_heights
array([ 5,  5,  0, 70])
>>> azi_bins.bin_locs
array([ 22.5,  67.5, 112.5, 157.5])
:rtype: :py:class:`~fractopo.analysis.azimuth.AzimuthBins`
```

```
>>> azi_bins.bin_width
45.0
```

fractopo.analysis.azimuth.**plot_azimuth_ax**(*bin_width*, *bin_locs*, *bin_heights*, *bar_color*, *ax*, *axial=True*)

Plot azimuth rose plot to ax.

fractopo.analysis.azimuth.**plot_azimuth_plot**(*azimuth_array*, *length_array*, *azimuth_set_array*, *azimuth_set_names*, *azimuth_set_ranges*, *label*, *plain*, *append_azimuth_set_text=False*, *add_abundance_order=False*, *axial=True*, *visualize_sets=False*, *bar_color='darkgrey'*)

Plot azimuth rose plot to its own figure.

Returns rose plot bin parameters, figure, ax

Return type
Tuple[*AzimuthBins*, Figure, PolarAxes]

fractopo.analysis.contour_grid module

Scripts for creating sample grids for fracture trace, branch and node data.

fractopo.analysis.contour_grid.**create_grid**(*cell_width*, *branches*)

Create an empty polygon grid for sampling fracture branch data.

Grid is created to always contain all given branches.

E.g.

```
>>> branches = gpd.GeoSeries(
...     [
...         LineString([(1, 1), (2, 2)]),
...         LineString([(2, 2), (3, 3)]),
...         LineString([(3, 0), (2, 2)]),
...         LineString([(2, 2), (-2, 5)]),
...     ]
... )
>>> create_grid(cell_width=0.1, branches=branches).head(5)
                                             geometry
0  POLYGON ((-2.00000 5.00000, -1.90000 5.00000, ...
1  POLYGON ((-2.00000 4.90000, -1.90000 4.90000, ...
2  POLYGON ((-2.00000 4.80000, -1.90000 4.80000, ...
:rtype: :py:class:`~geopandas.geodataframe.GeoDataFrame`
```

3 POLYGON ((-2.00000 4.70000, -1.90000 4.70000, … 4 POLYGON ((-2.00000 4.60000, -1.90000 4.60000, …

fractopo.analysis.contour_grid.**populate_sample_cell**(*sample_cell*, *sample_cell_area*, *traces*, *nodes*, *branches*, *snap_threshold*, *resolve_branches_and_nodes*, *traces_sindex=None*)

Take a single grid polygon and populate it with parameters.

Mauldon determination requires that E-nodes are defined for every single sample circle. If correct Mauldon values are wanted *resolve_branches_and_nodes* must be passed as True. This will result in much longer analysis time.

Return type
Dict[str, float]

fractopo.analysis.contour_grid.**sample_grid**(*grid*, *traces*, *nodes*, *branches*, *snap_threshold*, *resolve_branches_and_nodes=False*)

Populate a sample polygon grid with geometrical and topological parameters.

Return type
GeoDataFrame

fractopo.analysis.length_distributions module

Utilities for analyzing and plotting length distributions for line data.

class fractopo.analysis.length_distributions.**Dist**(*value*)

Bases: Enum

Enums of powerlaw model types.

EXPONENTIAL = 'exponential'

LOGNORMAL = 'lognormal'

POWERLAW = 'power_law'

TRUNCATED_POWERLAW = 'truncated_power_law'

class fractopo.analysis.length_distributions.**LengthDistribution**(*lengths*, *area_value*, *using_branches*, *name=''*, *_automatic_fit=None*)

Bases: object

Dataclass for length distributions.

area_value: float

property automatic_fit: Optional[Fit]

Get automatic powerlaw Fit.

generate_distributions(*cut_off=1e-18*)

Generate ccdf and truncated length data with cut_off.

lengths: ndarray

manual_fit(*cut_off*)

Get manual powerlaw Fit.

name: str = ''

using_branches: bool

class fractopo.analysis.length_distributions.**MultiLengthDistribution**(*distributions*, *using_branches*, *fitter=<function numpy_polyfit>*, *cut_offs=None*, *_fit_to_multi_scale_lengths=None*, *_normalized_distributions=None*, *_optimized=False*)

Bases: object

Multi length distribution.

cut_offs: Optional[List[float]] = None

distributions: List[*LengthDistribution*]

fitter(*log_ccm*)

Fit numpy polyfit to data.

Return type
Tuple[float, float]

property names: List[str]

Get length distribution names.

normalized_distributions(*automatic_cut_offs*)

Create normalized and truncated lengths and ccms.

Return type
Tuple[List[ndarray], List[ndarray], List[ndarray], List[ndarray], List[ndarray]]

optimize_cut_offs(*shgo_kwargs=None*, *scorer=<function mean_squared_log_error>*)

Get cut-off optimized MultiLengthDistribution.

Return type
Tuple[*MultiScaleOptimizationResult*, *MultiLengthDistribution*]

optimized_multi_scale_fit(*scorer*, *shgo_kwargs*)

Use scipy.optimize.shgo to optimize fit.

Return type
MultiScaleOptimizationResult

plot_multi_length_distributions(*automatic_cut_offs*, *plot_truncated_data*, *scorer=<function mean_squared_log_error>*)

Plot multi-scale length distribution.

Return type
Tuple[*Polyfit*, Figure, Axes]

using_branches: bool

class fractopo.analysis.length_distributions.**MultiScaleOptimizationResult**(*polyfit: Polyfit*, *cut_offs: ndarray*, *optimize_result: OptimizeResult*, *x0: ndarray*, *bounds: ndarray*, *proportions_of_data: List[float]*)

Bases: tuple

Results of scipy.optimize.shgo on length data.

bounds: ndarray

Alias for field number 4

cut_offs: ndarray

Alias for field number 1

optimize_result: OptimizeResult

Alias for field number 2

polyfit: *Polyfit*

Alias for field number 0

proportions_of_data: List[float]

Alias for field number 5

x0: ndarray

Alias for field number 3

class fractopo.analysis.length_distributions.**Polyfit**(*y_fit: ndarray*, *m_value: float*, *constant: float*, *score: float*, *scorer: Callable[[ndarray, ndarray], float]*)

Bases: tuple

Results of a polyfit to length data.

constant: float

Alias for field number 2

m_value: float

Alias for field number 1

score: float

Alias for field number 3

scorer: Callable[[ndarray, ndarray], float]

Alias for field number 4

y_fit: ndarray

Alias for field number 0

class fractopo.analysis.length_distributions.**SilentFit**(*data*, *discrete=False*, *xmin=None*, *xmax=None*, *verbose=True*, *fit_method='Likelihood'*, *estimate_discrete=True*, *discrete_approximation='round'*, *sigma_threshold=None*, *parameter_range=None*, *fit_optimizer=None*, *xmin_distance='D'*, *xmin_distribution='power_law'*, ***kwargs*)

Bases: `Fit`

Wrap powerlaw.Fit for the singular purpose of silencing output.

Silences output both to stdout and stderr.

fractopo.analysis.length_distributions.**all_fit_attributes_dict**(*fit*)

Collect 'all' fit attributes into a dict.

Return type
`Dict[str, float]`

fractopo.analysis.length_distributions.**apply_cut_off**(*lengths*, *ccm*, *cut_off=1e-18*)

Apply cut-off to length data and associated ccm.

```
>>> lengths = np.array([2, 4, 8, 16, 32])
>>> ccm = np.array([1. , 0.8, 0.6, 0.4, 0.2])
>>> cut_off = 4.5
:rtype: :py:data:`~typing.Tuple`\[:py:class:`~numpy.ndarray`, :py:class:`~numpy.
↪ndarray`]
```

```
>>> apply_cut_off(lengths, ccm, cut_off)
(array([ 8, 16, 32]), array([0.6, 0.4, 0.2]))
```

fractopo.analysis.length_distributions.**calculate_exponent**(*alpha*)

Calculate exponent from powerlaw.alpha.

fractopo.analysis.length_distributions.**calculate_fitted_values**(*log_lengths*, *m_value*, *constant*)

Calculate fitted values of y.

Return type
`ndarray`

fractopo.analysis.length_distributions.**cut_off_proportion_of_data**(*fit*, *length_array*)

Get the proportion of data cut off by *powerlaw* cut off.

If no fit is passed the cut off is the one used in *automatic_fit*.

Return type
`float`

fractopo.analysis.length_distributions.**describe_powerlaw_fit**(*fit*, *length_array*, *label=None*)

Compose dict of fit powerlaw attributes and comparisons between fits.

Return type
`Dict[str, float]`

fractopo.analysis.length_distributions.**distribution_compare_dict**(*fit*)

Compose a dict of length distribution fit comparisons.

Return type
`Dict[str, float]`

fractopo.analysis.length_distributions.**fit_to_multi_scale_lengths**(*ccm*, *lengths*, *fitter=<function numpy_polyfit>*, *scorer=<function mean_squared_log_error>*)

Fit np.polyfit to multiscale length distributions.

Returns the fitted values, exponent and constant of fit within a `Polyfit` instance.

Return type
Polyfit

fractopo.analysis.length_distributions.**numpy_polyfit**(*log_lengths*, *log_ccm*)

Fit numpy polyfit to data.

Return type
`Tuple[float, float]`

fractopo.analysis.length_distributions.**optimize_cut_offs**(*cut_offs*, *distributions*, *fitter*, *scorer*, **_*)

Optimize multiscale fit.

Requirements for the optimization function are that the function must take one argument of 1-d array and return a single float. It can take static arguments (distributions, fitter).

Return type
`float`

fractopo.analysis.length_distributions.**plot_distribution_fits**(*length_array*, *label*, *using_branches*, *use_probability_density_function*, *cut_off=None*, *fit=None*, *fig=None*, *ax=None*, *fits_to_plot=(<Dist.EXPONENTIAL: 'exponential'>, <Dist.LOGNORMAL: 'lognormal'>, <Dist.POWERLAW: 'power_law'>)*, *plain=False*)

Plot length distribution and *powerlaw* fits.

If a powerlaw.Fit is not given it will be automatically determined (using the optionally given cut_off).

Return type
Tuple[Optional[Fit], Figure, Axes]

fractopo.analysis.length_distributions.**plot_fit_on_ax**(*ax*, *fit*, *fit_distribution*, *use_probability_density_function*)

Plot powerlaw model to ax.

Return type
None

fractopo.analysis.length_distributions.**plot_multi_distributions_and_fit**(*truncated_length_array_all*, *ccm_array_normed_all*, *full_length_array_all*, *full_ccm_array_normed_all*, *cut_offs*, *names*, *polyfit*, *using_branches*, *plot_truncated_data*)

Plot multi-scale length distribution.

Return type
Tuple[Figure, Axes]

fractopo.analysis.length_distributions.**scikit_linear_regression**(*log_lengths*, *log_ccm*)

Fit using scikit LinearRegression.

Return type
Tuple[float, float]

fractopo.analysis.length_distributions.**setup_ax_for_ld**(*ax*, *using_branches*, *indiv_fit*, *use_probability_density_function*, *plain=False*)

Configure ax for length distribution plots.

Parameters

- **ax** (Axes) – Ax to setup.
- **using_branches** (bool) – Are the lines in the axis branches or traces.
- **indiv_fit** (bool) – Is the plot single-scale or multi-scale.
- **use_probability_density_function** (bool) – Whether to use complementary cumulative distribution function
- **plain** (bool) – Should the stylizing be kept to a minimum.

fractopo.analysis.length_distributions.**setup_length_dist_legend**(*ax*)

Set up legend for length distribution plots.

Used for both single and multi distribution plots.

fractopo.analysis.length_distributions.**sort_and_log_lengths_and_ccm**(*lengths*, *ccm*)

Preprocess lengths and ccm.

Sorts them and calculates their natural logarithmic.

Return type
Tuple[ndarray, ndarray]

fractopo.analysis.length_distributions.**sorted_lengths_and_ccm**(*lengths*, *area_value*)

Get (normalized) complementary cumulative number array.

Give area_value as None to **not** normalize.

```
>>> lengths = np.array([2, 4, 8, 16, 32])
>>> area_value = 10.0
:rtype: :py:data:`~typing.Tuple`\[:py:class:`~numpy.ndarray`, :py:class:`~numpy.
↪ndarray`]
```

```
>>> sorted_lengths_and_ccm(lengths, area_value)
(array([ 2,  4,  8, 16, 32]), array([0.1 , 0.08, 0.06, 0.04, 0.02]))
```

```
>>> lengths = np.array([2, 4, 8, 16, 32])
>>> area_value = None
>>> sorted_lengths_and_ccm(lengths, area_value)
(array([ 2,  4,  8, 16, 32]), array([1. , 0.8, 0.6, 0.4, 0.2]))
```

fractopo.analysis.line_data module

Trace and branch data analysis with LineData class abstraction.

class fractopo.analysis.line_data.**LineData**(*_line_gdf*, *using_branches*, *azimuth_set_ranges*, *azimuth_set_names*, *length_set_ranges=()*, *length_set_names=()*, *area_boundary_intersects=<factory>*, *_automatic_fit=None*)

Bases: `object`

Wrapper around the given GeoDataFrame with trace or branch data.

The line_gdf reference is passed and LineData will modify the input line_gdf instead of copying the input frame. This means line_gdf columns are accessible in the passed input reference upstream.

area_boundary_intersects: ndarray

property automatic_fit: Optional[Fit]

Get automatic powerlaw Fit.

property azimuth_array: ndarray

Array of trace or branch azimuths.

property azimuth_set_array: ndarray

Array of trace or branch azimuth set ids.

property azimuth_set_counts: Dict[str, int]

Get dictionary of azimuth set counts.

property azimuth_set_length_arrays: Dict[str, ndarray]

Get length arrays of each azimuth set.

azimuth_set_names: Tuple[str, ...]

azimuth_set_ranges: Tuple[Tuple[float, float], ...]

property boundary_intersect_count: Dict[str, int]

Get counts of line intersects with boundary.

boundary_intersect_count_desc(*label*)

Get counts of line intersects with boundary.

Return type
Dict[str, int]

describe_fit(*label=None*, *cut_off=None*)

Return short description of automatic powerlaw fit.

determine_manual_fit(*cut_off*)

Get manually determined Fit with set cut off.

Return type
Optional[Fit]

property geometry: GeoSeries

Get line geometries.

property length_array: ndarray

Array of trace or branch lengths.

Note: lengths can be 0.0 due to boundary weighting.

property length_array_non_weighted: ndarray

Array of trace or branch lengths not weighted by boundary conditions.

property length_boundary_weights

Array of weights for lines based on intersection count with boundary.

property length_set_array: ndarray

Array of trace or branch length set ids.

property length_set_counts: Dict[str, int]

Get dictionary of length set counts.

length_set_names: Tuple[str, ...] = ()

length_set_ranges: Tuple[Tuple[float, float], ...] = ()

plot_azimuth(*label*, *append_azimuth_set_text=False*, *add_abundance_order=False*, *visualize_sets=False*, *bar_color='darkgrey'*, *plain=False*)

Plot azimuth data in rose plot.

Return type
Tuple[*AzimuthBins*, Figure, PolarAxes]

plot_azimuth_set_count(*label*)

Plot azimuth set counts.

Return type
Tuple[Figure, Axes]

plot_azimuth_set_lengths(*use_probability_density_function=False*)

Plot azimuth set length distributions with fits.

Return type
Tuple[List[Optional[Fit]], List[Figure], List[Axes]]

plot_length_set_count(*label*)

Plot length set counts.

Return type
Tuple[Figure, Axes]

plot_lengths(*label*, *use_probability_density_function*, *fit=None*, *plain=False*)

Plot length data with powerlaw fit.

Return type
Tuple[Optional[Fit], Figure, Axes]

using_branches: bool

fractopo.analysis.multi_network module

MultiNetwork implementation for handling multiple network analysis.

class fractopo.analysis.multi_network.**MultiNetwork**(*networks: Tuple[*Network*, ...]*)

Bases: tuple

Multiple Network analysis.

basic_network_descriptions_df(*columns*)

Create DataFrame useful for basic Network characterization.

columns should contain key value pairs where the key is the column name in numerical_network_description dict. Value is a tuple where the first member is a new name for the column or alternatively None in which case the column name isn't changed. The second member should be the type of the column, typically either str, int or float.

collective_azimuth_sets()

Get collective azimuth set names.

Checks that all Networks have the same azimuth sets.

Return type
Tuple[Tuple[str, ...], Tuple[Tuple[float, float], ...]]

multi_length_distributions(*using_branches*)

Get MultiLengthDistribution of all networks.

Return type
MultiLengthDistribution

network_length_distributions(*using_branches*, *using_azimuth_sets*)

Get length distributions of Networks.

Return type
Dict[str, Dict[str, *LengthDistribution*]]

networks: Tuple[*Network*, ...]

Alias for field number 0

plot_branch(*colors=None*)

Plot multi-network ternary branch type plot.

plot_branch_azimuth_set_lengths(*automatic_cut_offs*, *plot_truncated_data*)

Plot multi-network trace azimuths set lengths with fits.

Return type
Tuple[Dict[str, *MultiLengthDistribution*], List[*Polyfit*], List[Figure], List[Axes]]

plot_multi_length_distribution(*using_branches*, *automatic_cut_offs*, *plot_truncated_data*, *multi_distribution=None*)

Plot multi-length distribution fit.

Use `multi_length_distributions()` to get most parameters used in fitting the multi-scale distribution.

Return type
Tuple[*MultiLengthDistribution*, *Polyfit*, Figure, Axes]

plot_trace_azimuth_set_lengths(*automatic_cut_offs*, *plot_truncated_data*)

Plot multi-network trace azimuths set lengths with fits.

Return type
Tuple[Dict[str, *MultiLengthDistribution*], List[*Polyfit*], List[Figure], List[Axes]]

plot_xyi(*colors=None*)

Plot multi-network ternary XYI plot.

set_multi_length_distributions(*using_branches*)

Get set-wise multi-length distributions.

Return type
Dict[str, *MultiLengthDistribution*]

subsample(*min_radii*, *random_choice=RandomChoice.radius*, *samples=1*)

Subsample all `Networks` in `MultiNetwork`.

Parameters

- **min_radii** (Union[float, Dict[str, float]]) – The minimum radius for subsamples can be either given as static or as a mapping that maps network names to specific radii.
- **random_choice** (*RandomChoice*) – Whether to use radius or area as the random choice parameter.
- **samples** (int) – How many subsamples to conduct per network.

Return type
List[Optional[Dict[str, Union[float, int, str]]]]

Returns
Subsamples

fractopo.analysis.network module

Analyse and plot trace map data with Network.

class fractopo.analysis.network.**Network**(*trace_gdf*, *area_gdf*, *name='Network'*, *determine_branches_nodes=False*, *snap_threshold=0.001*, *truncate_traces=True*, *circular_target_area=False*, *azimuth_set_names=('1', '2', '3')*, *azimuth_set_ranges=((0, 60), (60, 120), (120, 180))*, *trace_length_set_names=()*, *trace_length_set_ranges=()*, *branch_length_set_names=()*, *branch_length_set_ranges=()*, *branch_gdf=<factory>*, *node_gdf=<factory>*, *censoring_area=<factory>*, *cache_results=True*, *remove_z_coordinates_from_inputs=True*, *_anisotropy=None*, *_parameters=None*, *_azimuth_set_relationships=None*, *_trace_length_set_relationships=None*, *_trace_intersects_target_area_boundary=None*, *_branch_intersects_target_area_boundary=None*)

Bases: `object`

Trace network.

Consists of at its simplest of validated traces and a target area that delineates the traces i.e., only `trace_gdf` and `area_gdf` parameters are required to run the network analysis but results might not be correct or match your expectations e.g., traces are truncated to target area by default.

Parameters

- **trace_gdf** (`GeoDataFrame`) – `GeoDataFrame` containing trace data i.e. `shapely.geometry.LineString` geometries.
- **area_gdf** (`GeoDataFrame`) – `GeoDataFrame` containing target area data i.e. `(Multi)Polygon's`.
- **name** (`str`) – Name the Network.
- **determine_branches_nodes** (`bool`) – Whether to determine branches and nodes.
- **snap_threshold** (`float`) – The snapping distance threshold to identify snapped traces.
- **truncate_traces** (`bool`) – Whether to crop the traces at the target area boundary.
- **circular_target_area** (`bool`) – Is the target are a circle.
- **azimuth_set_names** (`Tuple[str, ...]`) – Names of each azimuth set.
- **azimuth_set_ranges** (`Tuple[Tuple[float, float], ...]`) – Ranges of each azimuth set.
- **trace_length_set_names** (`Tuple[str, ...]`) – Names of each trace length set.
- **trace_length_set_ranges** (`Tuple[Tuple[float, float], ...]`) – Ranges of each trace length set.
- **branch_length_set_names** (`Tuple[str, ...]`) – Names of each branch length set.
- **branch_length_set_ranges** (`Tuple[Tuple[float, float], ...]`) – Ranges of each branch length set.
- **branch_gdf** (`GeoDataFrame`) – `GeoDataFrame` containing branch data. It is recommended to let `fractopo.Network` determine both branches and nodes instead of passing them here.
- **node_gdf** (`GeoDataFrame`) – GeoDataFrame containing node data. It is recommended to let `fractopo.Network` determine both branches and nodes instead of passing them here.

- **censoring_area** (GeoDataFrame) – Polygon(s) in GeoDataFrame that delineate(s) the area in which trace digitization was uncertain due to censoring caused by e.g. vegetation.
- **cache_results** (bool) – Whether to use joblib memoize to disk-cache computationally expensive results.

property anisotropy: Tuple[ndarray, ndarray]

Determine anisotropy of connectivity.

area_gdf: GeoDataFrame

assign_branches_nodes(*branches=None*, *nodes=None*)

Determine and assign branches and nodes as attributes.

azimuth_set_names: Tuple[str, ...] = ('1', '2', '3')

azimuth_set_ranges: Tuple[Tuple[float, float], ...] = ((0, 60), (60, 120), (120, 180))

property azimuth_set_relationships: DataFrame

Determine azimuth set relationships.

property branch_azimuth_array: ndarray

Get branch azimuths as array.

property branch_azimuth_set_array: ndarray

Get azimuth set for each branch.

property branch_azimuth_set_counts: Dict[str, int]

Get branch azimuth set counts.

property branch_counts: Dict[str, int]

Get branch counts.

branch_gdf: GeoDataFrame

property branch_intersects_target_area_boundary: ndarray

Get array of E-component count.

property branch_length_array: ndarray

Get branch lengths as array.

property branch_length_array_non_weighted: ndarray

Get non-boundary-weighted branch lengths as array.

branch_length_distribution(*azimuth_set*)

Create structured LengthDistribution instance of branch length data.

Return type
LengthDistribution

property branch_length_set_array: ndarray

Get length set for each branch.

property branch_length_set_counts: Dict[str, int]

Get branch length set counts.

branch_length_set_names: Tuple[str, ...] = ()

branch_length_set_ranges: Tuple[Tuple[float, float], ...] = ()

branch_lengths_powerlaw_fit(*cut_off=None*)

Determine powerlaw fit for branch lengths.

Return type
Optional[Fit]

property branch_lengths_powerlaw_fit_description: Dict[str, float]

Short numerical description dict of branch length powerlaw fit.

property branch_series: GeoSeries

Get branch geometries as GeoSeries.

property branch_types: ndarray

Get branch type of each branch.

cache_results: bool = True

censoring_area: GeoDataFrame

circular_target_area: bool = False

contour_grid(*cell_width=None*, *bounds_divider=20.0*, *precursor_grid=None*, *resolve_branches_nodes=False*)

Sample the network with a contour grid.

If `cell_width` is passed it is used as the cell width. Otherwise a cell width is calculated using the network branch bounds using the passed `bounds_divider` or its default value.

If `precursor_grid` is passed it is used as the grid in which each Polygon cell is filled with calculated network parameter values.

determine_branches_nodes: bool = False

estimate_censoring()

Estimate the amount of censoring as area float value.

Censoring is caused by e.g. vegetation.

Returns np.nan if no `censoring_area` is passed by the user into `Network` creation or if the passed GeoDataFrame is empty.

Return type
float

export_network_analysis(*output_path*, *include_contour_grid=True*)

Export pre-selected `Network` analysis results to a directory.

The chosen analyses are opionated but should contain at least the basic results of fracture network analysis.

`output_path` should correspond to a path to an existing or directory or direct path to a non-existing directory where one will be created.

property length_set_relationships: DataFrame

Determine length set relationships.

name: str = 'Network'

property node_counts: Dict[str, int]

Get node counts.

node_gdf: GeoDataFrame

property node_series: GeoSeries

Get node geometries as GeoSeries.

property node_types: ndarray

Get node type of each node.

numerical_network_description(*trace_lengths_cut_off=None*, *branch_lengths_cut_off=None*)

Collect numerical network attributes and return them as a dictionary.

Return type
Dict[str, Union[float, int, str]]

property parameters: Dict[str, float]

Get numerical geometric and topological parameters.

property plain_name

Get filename friendly name for Network based on name attribute.

plot_anisotropy(*label=None*, *color=None*)

Plot anisotropy of connectivity plot.

Return type
Optional[Tuple[Figure, Axes]]

plot_azimuth_crosscut_abutting_relationships()

Plot azimuth set crosscutting and abutting relationships.

Return type
Tuple[List[Figure], List[ndarray]]

plot_branch(*label=None*)

Plot ternary plot of branch types.

Return type
Tuple[Figure, Axes, TernaryAxesSubplot]

plot_branch_azimuth(*label=None*, *append_azimuth_set_text=False*, *add_abundance_order=False*, *visualize_sets=False*, *bar_color='darkgrey'*, *plain=False*)

Plot branch azimuth rose plot.

Return type
Tuple[*AzimuthBins*, Figure, PolarAxes]

plot_branch_azimuth_set_count(*label=None*)

Plot branch azimuth set counts.

Return type
Tuple[Figure, Axes]

plot_branch_azimuth_set_lengths()

Plot branch azimuth set lengths with fits.

Return type
Tuple[List[Optional[Fit]], List[Figure], List[Axes]]

plot_branch_length_set_count(*label=None*)

Plot branch length set counts.

Return type
Tuple[Figure, Axes]

plot_branch_lengths(*label=None*, *fit=None*, *use_probability_density_function=False*, *plain=False*)

Plot branch length distribution with *powerlaw* fits.

Return type
Tuple[Optional[Fit], Figure, Axes]

plot_contour(*parameter*, *sampled_grid*)

Plot contour plot of a geometric or topological parameter.

Creating the contour grid is expensive so the `sampled_grid` must be first created with `Network.contour_grid` method and then passed to this one for plotting.

Return type
Tuple[Figure, Axes]

plot_parameters(*label=None*, *color=None*)

Plot geometric and topological parameters.

Return type
Optional[Tuple[Figure, Axes]]

plot_trace_azimuth(*label=None*, *append_azimuth_set_text=False*, *add_abundance_order=False*, *visualize_sets=False*, *bar_color='darkgrey'*, *plain=False*)

Plot trace azimuth rose plot.

Return type
Tuple[*AzimuthBins*, Figure, PolarAxes]

plot_trace_azimuth_set_count(*label=None*)

Plot trace azimuth set counts.

Return type
Tuple[Figure, Axes]

plot_trace_azimuth_set_lengths()

Plot trace azimuth set lengths with fits.

Return type
Tuple[List[Optional[Fit]], List[Figure], List[Axes]]

plot_trace_length_crosscut_abutting_relationships()

Plot length set crosscutting and abutting relationships.

Return type
Tuple[List[Figure], List[ndarray]]

plot_trace_length_set_count(*label=None*)

Plot trace length set counts.

Return type
Tuple[Figure, Axes]

plot_trace_lengths(*label=None*, *fit=None*, *use_probability_density_function=False*, *plain=False*)

Plot trace length distribution with *powerlaw* fits.

Return type
Tuple[Fit, Figure, Axes]

plot_xyi(*label=None*)

Plot ternary plot of node types.

Return type
Tuple[Figure, Axes, TernaryAxesSubplot]

remove_z_coordinates_from_inputs: bool = True

representative_points()

Get representative point(s) of target area(s).

Return type
List[Point]

reset_length_data()

Reset LineData attributes.

WARNING: Mostly untested.

snap_threshold: float = 0.001

property target_areas: List[Union[Polygon, MultiPolygon]]

Get all target areas from area_gdf.

property total_area: float

Get total area.

property trace_azimuth_array: ndarray

Get trace azimuths as array.

property trace_azimuth_set_array: ndarray

Get azimuth set for each trace.

property trace_azimuth_set_counts: Dict[str, int]

Get trace azimuth set counts.

trace_gdf: GeoDataFrame

property trace_intersects_target_area_boundary: ndarray

Check traces for intersection with target area boundaries.

Results are in integers:

- 0 == No intersections
- 1 == One intersection
- 2 == Two intersections

Does not discriminate between which target area (if multiple) the trace intersects. Intersection detection based on snap_threshold.

property trace_length_array: ndarray

Get trace lengths as array.

property trace_length_array_non_weighted: ndarray

Get non-boundary-weighted trace lengths as array.

trace_length_distribution(*azimuth_set*)

Create structured LengthDistribution instance of trace length data.

Return type
LengthDistribution

property trace_length_set_array: ndarray

Get length set for each trace.

property trace_length_set_counts: Dict[str, int]

Get trace length set counts.

trace_length_set_names: Tuple[str, ...] = ()

trace_length_set_ranges: Tuple[Tuple[float, float], ...] = ()

trace_lengths_powerlaw_fit(*cut_off=None*)

Determine powerlaw fit for trace lengths.

Return type
Optional[Fit]

property trace_lengths_powerlaw_fit_description: Dict[str, float]

Short numerical description dict of trace length powerlaw fit.

property trace_series: GeoSeries

Get trace geometries as GeoSeries.

truncate_traces: bool = True

write_branches_and_nodes(*output_dir_path*, *branches_name=None*, *nodes_name=None*)

Write branches and nodes to disk.

Enables reuse of the same data in analysis of the same data to skip topology determination which is computationally expensive.

Writes only with the GeoJSON driver as there are differences between different spatial filetypes. Only GeoJSON is supported to avoid unexpected errors.

fractopo.analysis.network.**requires_topology**(*func*)

Wrap methods that require determined topology.

Raises an error if trying to call them without determined topology.

Return type
Callable

fractopo.analysis.parameters module

Analysis and plotting of geometric and topological parameters.

fractopo.analysis.parameters.**branches_intersect_boundary**(*branch_types*)

Get array of if branches have E-component (intersects target area).

Return type
ndarray

fractopo.analysis.parameters.**convert_counts**(*counts*)

Convert float and int value in counts to ints only.

Used by branches and nodes count calculators.

Return type
Dict[str, int]

fractopo.analysis.parameters.**counts_to_point**(*counts*, *is_nodes*, *scale=100*)

Create ternary point from node_counts.

The order is important: for nodes: X, I, Y and for branches: CC, II, CI.

Return type
Optional[Tuple[float, float, float]]

fractopo.analysis.parameters.**decorate_branch_ax**(*ax*, *tax*, *counts*)

Decorate ternary branch plot.

fractopo.analysis.parameters.**decorate_count_ax**(*ax*, *tax*, *label_counts*, *is_nodes*)

Decorate ternary count plot.

fractopo.analysis.parameters.**decorate_xyi_ax**(*ax*, *tax*, *counts*)

Decorate xyi plot.

fractopo.analysis.parameters.**determine_branch_type_counts**(*branch_types*, *branches_defined*)

Determine branch type counts.

Return type
Dict[str, Union[float, int]]

fractopo.analysis.parameters.**determine_node_type_counts**(*node_types*, *branches_defined*)

Determine node type counts.

Return type
Dict[str, Union[float, int]]

fractopo.analysis.parameters.**determine_set_counts**(*set_names*, *set_array*)

Determine counts in for each set.

Return type
Dict[str, int]

fractopo.analysis.parameters.**determine_topology_parameters**(*trace_length_array*, *area*, *branches_defined*, *correct_mauldon*, *node_counts*, *branch_length_array*)

Determine geometric (and topological) parameters.

Number of traces (and branches) are determined by node counting.

The passed `trace_length_array` should be non-weighted.

Return type
Dict[str, float]

fractopo.analysis.parameters.**initialize_ternary_ax**(*ax*, *tax*)

Decorate ternary ax for both XYI and branch types.

Returns a font dict for use in plotting text.

Return type
dict

fractopo.analysis.parameters.**initialize_ternary_branches_points**(*ax*, *tax*)

Initialize ternary branches plot ax and tax.

fractopo.analysis.parameters.**initialize_ternary_points**(*ax*, *tax*)

Initialize ternary points figure ax and tax.

fractopo.analysis.parameters.**plot_branch_plot_ax**(*counts*, *label*, *tax*, *color=None*)

Plot ternary branch plot to tax.

fractopo.analysis.parameters.**plot_parameters_plot**(*topology_parameters_list*, *labels*, *colors=None*)

Plot topological parameters.

fractopo.analysis.parameters.**plot_set_count**(*set_counts*, *label*)

Plot set counts.

Return type
Tuple[Figure, Axes]

fractopo.analysis.parameters.**plot_ternary_plot**(*counts_list*, *labels*, *is_nodes*, *colors=None*)

Plot ternary plot.

Same function is used to plot both XYI and branch type ternary plots.

Return type
Tuple[Figure, Axes, TernaryAxesSubplot]

fractopo.analysis.parameters.**plot_xyi_plot_ax**(*counts*, *label*, *tax*, *color=None*)

Plot XYI pointst to given ternary axis (tax).

fractopo.analysis.parameters.**tern_plot_branch_lines**(*tax*)

Plot line of random assignment of nodes to branches to a branch ternary tax.

Line positions taken from NetworkGT open source code. Credit to: https://github.com/BjornNyberg/NetworkGT

Parameters
tax (*ternary.TernaryAxesSubplot*) – Ternary axis to plot to

fractopo.analysis.parameters.**tern_plot_the_fing_lines**(*tax*, *cs_locs=(1.3, 1.5, 1.7, 1.9)*)

Plot *connections per branch* parameter to XYI-plot.

If not using the pre-determined locations the texts will not be correctly placed as they use absolute positions and labels.

Parameters

- **tax** (TernaryAxesSubplot) – Ternary axis to plot to.
- **cs_locs** – Pre-determined locations for the lines.

fractopo.analysis.parameters.**tern_yi_func**(*c*, *x*)

Plot *Connections per branch* threshold line to branch ternary plot.

Uses absolute values.

fractopo.analysis.parameters.**ternary_heatmapping**(*x_values*, *y_values*, *i_values*, *number_of_bins*, *scale_divider=1.0*, *ax=None*)

Plot ternary heatmap.

Modified from: https://github.com/marcharper/python-ternary/issues/81

Return type
Tuple[Figure, TernaryAxesSubplot]

fractopo.analysis.parameters.**ternary_point_kwargs**(*alpha=1.0*, *zorder=4*, *s=25*, *marker='X'*)

Plot point to a ternary figure.

fractopo.analysis.parameters.**ternary_text**(*text*, *ax*)

Add ternary text about counts.

fractopo.analysis.random_sampling module

Utilities for randomly Network sampling traces.

class fractopo.analysis.random_sampling.**NetworkRandomSampler**(*trace_gdf*, *area_gdf*, *min_radius*, *snap_threshold*, *random_choice*, *name*)

Bases: object

Randomly sample traces inside given target area.

area_gdf: GeoDataFrame

static area_gdf_should_contain_polygon(*area_gdf*)

Check that area_gdf contains one Polygon.

Return type
GeoDataFrame

property max_area: float

Calculate maximum area from max_radius.

property max_radius: float

Calculate max radius from given area_gdf.

property min_area: float

Calculate minimum area from min_radius.

min_radius: float

name: str

random_area()

Calculate random area in area range.

Range is calculated from [min_radius, max_radius[.

Return type
float

random_choice: Union[*RandomChoice*, str]

static random_choice_should_be_enum(*random_choice*)

Check that random_choice is valid.

Return type
RandomChoice

random_network_sample(*determine_branches_nodes=True*)

Get random Network sample with a random target area.

Returns the network, the sample circle centroid and circle radius.

Return type
RandomSample

classmethod random_network_sampler(*network*, *min_radius*, *random_choice=RandomChoice.radius*)

Initialize `NetworkRandomSampler` for random sampling.

Assumes that `Network` target area is a single `Polygon` circle.

random_radius()

Calculate random radius in range [min_radius, max_radius[.

Return type
`float`

random_target_circle()

Get random target area and its centroid and radius.

The target area is always within the original target area.

Return type
`Tuple[Polygon, Point, float]`

snap_threshold: float

property target_area_centroid: Point

Get target area centroid.

property target_circle: Polygon

Target circle Polygon from area_gdf.

trace_gdf: GeoDataFrame

static trace_gdf_should_contain_traces(*trace_gdf*)

Check that trace_gdf contains LineString traces.

Return type
`GeoDataFrame`

static value_should_be_positive(*min_radius*)

Check that value is positive.

Return type
`float`

class fractopo.analysis.random_sampling.**RandomChoice**(*value*)

Bases: `Enum`

Choose between random area or radius.

The choice is relevant because area is polynomially correlated with radius.

area = 'area'

radius = 'radius'

class fractopo.analysis.random_sampling.**RandomSample**(*network_maybe*, *target_centroid*, *radius*, *name*)

Bases: object

Dataclass for sampling results.

name: str

network_maybe: Optional[*Network*]

radius: float

target_centroid: Point

fractopo.analysis.relationships module

Functions for plotting cross-cutting and abutting relationships.

fractopo.analysis.relationships.**determine_crosscut_abutting_relationships**(*trace_series*, *node_series*, *node_types*, *set_array*, *set_names*, *buffer_value*, *label*)

Determine cross-cutting and abutting relationships between trace sets.

Determines relationships between all inputted sets by using spatial intersects between node and trace data.

E.g.

```
>>> trace_series = gpd.GeoSeries(
...     [LineString([(0, 0), (1, 0)]), LineString([(0, 1), (0, -1)])]
... )
>>> node_series = gpd.GeoSeries(
...     [Point(0, 0), Point(1, 0), Point(0, 1), Point(0, -1)]
... )
>>> node_types = np.array(["Y", "I", "I", "I"])
>>> set_array = np.array(["1", "2"])
>>> set_names = ("1", "2")
>>> buffer_value = 0.001
>>> label = "title"
>>> determine_crosscut_abutting_relationships(
...     trace_series,
...     node_series,
...     node_types,
...     set_array,
...     set_names,
...     buffer_value,
...     label,
... )
:rtype: :py:class:`~pandas.core.frame.DataFrame`
```

name sets x y y-reverse error-count

0 title (1, 2) 0 1 0 0

TODO: No within set relations…..yet… Problem?

fractopo.analysis.relationships.**determine_intersect**(*node*, *node_class*, *l1*, *l2*, *first_set*, *second_set*, *first_setpointtree*, *buffer_value*)

Determine what intersection the node represents.

TODO: R0912: Too many branches.

Return type
Dict[str, Union[Point, str, Tuple[str, str], bool]]

fractopo.analysis.relationships.**determine_intersects**(*trace_series_two_sets*, *set_names_two_sets*, *node_series_xy_intersects*, *node_types_xy_intersects*, *buffer_value*)

Determine how abutments and crosscuts occur between two sets.

E.g.

```
>>> traces = gpd.GeoSeries([LineString([(0, 0), (1, 1)])]), gpd.GeoSeries(
...     [LineString([(0, 1), (0, -1)])]
... )
>>> set_names_two_sets = ("1", "2")
>>> node_series_xy_intersects = gpd.GeoSeries([Point(0, 0)])
>>> node_types_xy_intersects = np.array(["Y"])
>>> buffer_value = 0.001
>>> determine_intersects(
...     traces,
...     set_names_two_sets,
...     node_series_xy_intersects,
...     node_types_xy_intersects,
...     buffer_value,
... )
:rtype: :py:class:`~pandas.core.frame.DataFrame`
```

node nodeclass sets error

0 POINT (0 0) Y (1, 2) False

fractopo.analysis.relationships.**determine_nodes_intersecting_sets**(*trace_series_two_sets*, *set_names_two_sets*, *node_series_xy*, *buffer_value*)

Conduct a spatial intersect between nodes and traces.

Node GeoDataFrame contains only X- and Y-nodes and the trace GeoDataFrame only two sets. Returns boolean array of based on intersections.

E.g.

```
>>> traces = gpd.GeoSeries([LineString([(0, 0), (1, 1)])]), gpd.GeoSeries(
...     [LineString([(0, 1), (0, -1)])]
... )
>>> set_names_two_sets = ("1", "2")
>>> nodes_xy = gpd.GeoSeries([Point(0, 0), Point(1, 1), Point(0, 1), Point(0, -1)])
>>> buffer_value = 0.001
>>> determine_nodes_intersecting_sets(
...     traces, set_names_two_sets, nodes_xy, buffer_value
:rtype: :py:class:`~typing.List`\[:py:class:`bool`]
```

...) [True, False, False, False]

fractopo.analysis.relationships.**plot_crosscut_abutting_relationships_plot**(*relations_df*, *set_array*, *set_names*)

Plot cross-cutting and abutting relationships.

Return type
Tuple[List[Figure], List[ndarray]]

fractopo.analysis.subsampling module

Utilities for Network subsampling.

fractopo.analysis.subsampling.**aggregate_chosen**(*chosen*, *default_aggregator=<function mean_aggregation>*)

Aggregate a collection of subsampled circles for params.

Weights averages by the area of each subsampled circle.

Return type
Dict[str, Any]

fractopo.analysis.subsampling.**area_weighted_index_choice**(*idxs*, *areas*, *compressor*)

Make area-weighted choce from list of indexes.

Return type
int

fractopo.analysis.subsampling.**choose_sample_from_group**(*group*)

Choose single sample from group DataFrame.

Return type
Dict[str, Union[float, int, str]]

fractopo.analysis.subsampling.**collect_indexes_of_base_circles**(*idxs*, *how_many*, *areas*)

Collect indexes of base circles, area-weighted and randomly.

Return type
List[int]

fractopo.analysis.subsampling.**create_sample**(*sampler*)

Sample with NetworkRandomSampler and return Network description.

Return type
Optional[Dict[str, Union[float, int, str]]]

fractopo.analysis.subsampling.**gather_subsample_descriptions**(*subsample_results*)

Gather results from a list of subsampling ProcessResults.

Return type
List[Dict[str, Union[float, int, str]]]

fractopo.analysis.subsampling.**group_gathered_subsamples**(*subsamples*, *groupby_column='Name'*)

Group gathered subsamples.

By default groups by Name column.

```
>>> subsamples = [
...     {"param": 2.0, "Name": "myname"},
...     {"param": 2.0, "Name": "myname"},
... ]
:rtype: :py:class:`~typing.Dict`\[:py:class:`str`, :py:class:`~typing.List`\[:py:
↪class:`~typing.Dict`\[:py:class:`str`, :py:data:`~typing.Union`\[:py:class:
↪`float`, :py:class:`int`, :py:class:`str`]]]]
```

```
>>> group_gathered_subsamples(subsamples)
{'myname': [{'param': 2.0, 'Name': 'myname'}, {'param': 2.0, 'Name': 'myname'}]}
```

fractopo.analysis.subsampling.**groupby_keyfunc**(*item*, *groupby_column='Name'*)

Use groupby_column to group values.

Return type
str

fractopo.analysis.subsampling.**random_sample_of_circles**(*grouped*, *circle_names_with_diameter*, *min_circles=1*, *max_circles=None*)

Get a random sample of circles from grouped subsampled data.

Both the amount of overall circles and which circles within each group is random. Data is grouped by target area name.

Return type
List[Dict[str, Union[float, int, str]]]

fractopo.analysis.subsampling.**subsample_networks**(*networks*, *min_radii*, *random_choice=RandomChoice.radius*, *samples=1*)

Subsample given Sequence of Networks.

Return type
List[Optional[Dict[str, Union[float, int, str]]]]

Module contents

Contains most analysis utilities and abstractions.

Network class is defined in ./network.py and MultiNetwork in ./multi_network.py.

fractopo.tval package

Submodules

fractopo.tval.proximal_traces module

Determine traces that could be integrated together.

determine_proximal_traces takes an input of GeoSeries or GeoDataFrame of LineString geometries and returns a GeoDataFrame with a new column *Merge* which has values of True or False depending on if nearby proximal traces were found.

fractopo.tval.proximal_traces.**determine_proximal_traces**(*traces*, *buffer_value*, *azimuth_tolerance*)

Determine proximal traces.

Takes an input of GeoSeries or GeoDataFrame of LineString geometries and returns a GeoDataFrame with a new column *Merge* which has values of True or False depending on if nearby proximal traces were found.

E.g.

```
>>> lines = [
...     LineString([(0, 0), (0, 3)]),
...     LineString([(1, 0), (1, 3)]),
...     LineString([(5, 0), (5, 3)]),
...     LineString([(0, 0), (-3, -3)]),
... ]
>>> traces = gpd.GeoDataFrame({"geometry": lines})
>>> buffer_value = 1.1
>>> azimuth_tolerance = 10
>>> determine_proximal_traces(traces, buffer_value, azimuth_tolerance)
                                        geometry  Merge
0    LINESTRING (0.00000 0.00000, 0.00000 3.00000)   True
1    LINESTRING (1.00000 0.00000, 1.00000 3.00000)   True
:rtype: :py:class:`~geopandas.geodataframe.GeoDataFrame`
```

2 LINESTRING (5.00000 0.00000, 5.00000 3.00000) False 3 LINESTRING (0.00000 0.00000, -3.00000 -3.00000) False

fractopo.tval.proximal_traces.**is_similar_azimuth**(*trace*, *other*, *tolerance*)

Determine if azimuths of *trace* and *other* are close.

Checks both the start – end -azimuth and regression-based azimuth.

E.g.

```
>>> trace = LineString([(0, 0), (0, 3)])
>>> other = LineString([(0, 0), (0, 4)])
>>> is_similar_azimuth(trace, other, 1)
True
```

```
>>> trace = LineString([(0, 0), (1, 1)])
>>> other = LineString([(0, 0), (0, 4)])
>>> is_similar_azimuth(trace, other, 40)
False
>>> is_similar_azimuth(trace, other, 50)
True
>>> is_similar_azimuth(trace, other, 45)
False
```

fractopo.tval.proximal_traces.**is_within_buffer_distance**(*trace*, *other*, *buffer_value*)

Determine if *trace* and *other* are within buffer distance.

Threshold distance is buffer_value (both are buffered with half of buffer_value) of each other.

E.g.

```
>>> trace = LineString([(0, 0), (0, 3)])
>>> other = LineString([(0, 0), (0, 3)])
```

(continues on next page)

(continued from previous page)

```
>>> is_within_buffer_distance(trace, other, 0.1)
True
```

```
>>> line = LineString([(0, 0), (0, 3)])
>>> other = LineString([(3, 0), (3, 3)])
>>> is_within_buffer_distance(trace, other, 2)
False
>>> is_within_buffer_distance(trace, other, 4)
True
```

fractopo.tval.trace_validation module

Contains main entrypoint class for validating trace data, Validation.

Create Validation objects from traces and their target areas to validate the traces for further analysis (`fractopo.analysis.network.Network`).

class fractopo.tval.trace_validation.**Validation**(*traces*, *area*, *name*, *allow_fix*, *SNAP_THRESHOLD=0.01*, *SNAP_THRESHOLD_ERROR_MULTIPLIER=1.1*, *AREA_EDGE_SNAP_MULTIPLIER=1.5*, *TRIANGLE_ERROR_SNAP_MULTIPLIER=10.0*, *OVERLAP_DETECTION_MULTIPLIER=50.0*, *SHARP_AVG_THRESHOLD=135.0*, *SHARP_PREV_SEG_THRESHOLD=100.0*, *ERROR_COLUMN='VALIDATION_ERRORS'*, *GEOMETRY_COLUMN='geometry'*, *determine_validation_nodes=True*)

Bases: `object`

Validate traces data delineated by target area(s).

If allow_fix is True, some automatic fixing will be done to e.g. convert MultiLineStrings to LineStrings.

AREA_EDGE_SNAP_MULTIPLIER: float = 1.5

ERROR_COLUMN: str = 'VALIDATION_ERRORS'

GEOMETRY_COLUMN: str = 'geometry'

OVERLAP_DETECTION_MULTIPLIER: float = 50.0

SHARP_AVG_THRESHOLD: float = 135.0

SHARP_PREV_SEG_THRESHOLD: float = 100.0

SNAP_THRESHOLD: float = 0.01

SNAP_THRESHOLD_ERROR_MULTIPLIER: float = 1.1

TRIANGLE_ERROR_SNAP_MULTIPLIER: float = 10.0

allow_fix: bool

area: GeoDataFrame

determine_validation_nodes: bool = True

property endpoint_nodes: List[Tuple[Point, ...]]

Get endpoints of all traces.

Returned as a list of tuples wherein each tuple represents the nodes of a trace in traces i.e. endpoint_nodes[index] are the nodes for traces[index].

property faulty_junctions: Optional[Set[int]]

Determine indexes with Multi Junctions.

property intersect_nodes: List[Tuple[Point, ...]]

Get intersection nodes of all traces.

Returned as a list of tuples wherein each tuple represents the nodes of a trace in traces i.e. intersect_nodes[index] are the nodes for traces[index].

name: str

run_validation(*first_pass=True*, *choose_validators=None*, *allow_empty_area=True*)

Run validation.

Main entrypoint for validation. Returns validated traces GeoDataFrame.

Return type
GeoDataFrame

set_general_nodes()

Set _intersect_nodes and _endpoint_nodes attributes.

property spatial_index: Optional[PyGEOSSTRTreeIndex]

Get geopandas/pygeos spatial_index of traces.

traces: GeoDataFrame

property vnodes: Optional[Set[int]]

Determine indexes with V-Nodes.

fractopo.tval.trace_validation_utils module

Direct utilities of trace validation.

fractopo.tval.trace_validation_utils.**determine_middle_in_triangle**(*segments*, *snap_threshold*, *snap_threshold_error_multiplier*)

Determine the middle segment within a triangle error.

The middle segment always intersects the other two.

Return type
List[LineString]

fractopo.tval.trace_validation_utils.**determine_trace_candidates**(*geom*, *idx*, *traces*, *spatial_index*)

Determine potentially intersecting traces with spatial index.

Return type
GeoSeries

fractopo.tval.trace_validation_utils.**is_underlapping**(*geom*, *trace*, *endpoint*, *snap_threshold*, *snap_threshold_error_multiplier*)

Determine if a geom is underlapping.

Return type
Optional[bool]

fractopo.tval.trace_validation_utils.**linestring_segment**(*linestring*, *dist*, *threshold_length*)

Get LineString segment from dist to dist + threshold_length.

fractopo.tval.trace_validation_utils.**segment_within_buffer**(*linestring*, *multilinestring*, *snap_threshold*, *snap_threshold_error_multiplier*, *overlap_detection_multiplier*)

Check if segment is within buffer of multilinestring.

First check if given linestring completely overlaps any part of multilinestring and if it does, returns True.

Next it starts to segmentize the multilinestring to smaller linestrings and consequently checks if these segments are completely within a buffer made of the given linestring. It also checks that the segment size is reasonable.

TODO: segmentize_linestring is very inefficient.

Return type
bool

fractopo.tval.trace_validation_utils.**segmentize_linestring**(*linestring*, *threshold_length*)

Segmentize LineString to smaller parts.

Resulting parts are not guaranteed to be mergeable back to the original.

Return type
List[Tuple[Tuple[float, float], Tuple[float, float]]]

fractopo.tval.trace_validation_utils.**split_to_determine_triangle_errors**(*trace*, *splitter_trace*, *snap_threshold*, *triangle_error_snap_multiplier*)

Split trace with splitter_trace to determine triangle intersections.

Return type
bool

fractopo.tval.trace_validators module

Contains Validator classes which each have their own error to handle and mark.

class fractopo.tval.trace_validators.**BaseValidator**

Bases: object

Base validator that all classes inherit.

TODO: Validation requires overhaul at some point.

ERROR = 'BASE ERROR'

INTERACTION_NODES_COLUMN = 'IP'

LINESTRING_ONLY = True

static fix_method(**_)

Abstract fix method.

Return type
Optional[LineString]

static validation_method(**_)

Abstract validation method.

Return type
bool

class fractopo.tval.trace_validators.**EmptyTargetAreaValidator**

Bases: *BaseValidator*

Stub validator for empty target area.

Validation for this error occurs at the start of run_validation.

ERROR = 'EMPTY TARGET AREA'

class fractopo.tval.trace_validators.**GeomNullValidator**

Bases: *BaseValidator*

Validate the geometry for NULL GEOMETRY errors.

ERROR = 'NULL GEOMETRY'

LINESTRING_ONLY = False

static validation_method(*geom*, **_)

Validate for empty and null geometries.

E.g. some validations are handled by GeomTypeValidator

Return type
bool

```
>>> GeomNullValidator.validation_method(Point(1, 1))
True
```

Empty geometries are not valid.

```
>>> GeomNullValidator.validation_method(LineString())
False
```

```
>>> GeomNullValidator.validation_method(LineString([(-1, 1), (1, 1)]))
True
```

class fractopo.tval.trace_validators.**GeomTypeValidator**

Bases: *BaseValidator*

Validates the geometry type.

Validates that all traces are LineStrings. Tries to use shapely.ops.linemerge to merge MultiLineStrings into LineStrings.

ERROR = 'GEOM TYPE MULTILINESTRING'

LINESTRING_ONLY = False

static fix_method(*geom*, **_)

Fix mergeable MultiLineStrings to LineStrings.

E.g. mergeable MultiLineString

```
>>> mls = MultiLineString([((0, 0), (1, 1)), ((1, 1), (2, 2))])
:rtype: :py:data:`~typing.Optional`\[:py:class:`~shapely.geometry.linestring.
↪LineString`]
```

```
>>> GeomTypeValidator.fix_method(geom=mls).wkt
'LINESTRING (0 0, 1 1, 2 2)'
```

Unhandled types will just return as None.

```
>>> GeomTypeValidator.fix_method(geom=Point(1, 1)) is None
True
```

static validation_method(*geom*, **_)

Validate geometries.

E.g. Anything but LineString

Return type
bool

```
>>> GeomTypeValidator.validation_method(Point(1, 1))
False
```

With LineString:

```
>>> GeomTypeValidator.validation_method(LineString([(0, 0), (1, 1)]))
True
```

class fractopo.tval.trace_validators.**MultiJunctionValidator**

Bases: *BaseValidator*

Validates that junctions consists of a maximum of two lines crossing.

ERROR = 'MULTI JUNCTION'

static determine_faulty_junctions(*all_nodes*, *snap_threshold*, *snap_threshold_error_multiplier*)

Determine when a point of interest represents a multi junction.

I.e. when there are more than 2 points within the buffer distance of each other.

Two points is the limit because an intersection point between two traces is added to the list of interest points twice, once for each trace.

Faulty junction can also represent a slightly overlapping trace i.e. a snapping error.

```
>>> all_nodes = [
...     (Point(0, 0), Point(1, 1)),
...     (Point(1, 1),),
...     (Point(5, 5),),
...     (Point(0, 0), Point(1, 1)),
... ]
>>> snap_threshold = 0.01
```

(continues on next page)

(continued from previous page)

```
>>> snap_threshold_error_multiplier = 1.1
>>> MultiJunctionValidator.determine_faulty_junctions(
...     all_nodes, snap_threshold, snap_threshold_error_multiplier
:rtype: :py:class:`~typing.Set`\[:py:class:`int`]
```

...) {0, 1, 3}

static validation_method(*idx*, *faulty_junctions*, **_)

Validate for multi junctions between traces.

```
>>> MultiJunctionValidator.validation_method(
...     idx=1, faulty_junctions=set([1, 2])
:rtype: :py:class:`bool`
```

...) False

class fractopo.tval.trace_validators.**MultipleCrosscutValidator**

Bases: *BaseValidator*

Find traces that cross-cut each other multiple times.

This also indicates the possibility of duplicate traces.

ERROR = 'MULTIPLE CROSSCUTS'

static validation_method(*geom*, *trace_candidates*, **_)

Validate for multiple crosscuts.

```
>>> geom = LineString([Point(-3, -4), Point(-3, -1)])
>>> trace_candidates = gpd.GeoSeries(
...     [
...         LineString(
...             [Point(-4, -3), Point(-2, -3), Point(-4, -2), Point(-2, -1)]
...         )
...     ]
... )
:rtype: :py:class:`bool`
```

```
>>> MultipleCrosscutValidator.validation_method(geom, trace_candidates)
False
```

```
>>> geom = LineString([Point(-3, -4), Point(-3, -4.5)])
>>> trace_candidates = gpd.GeoSeries(
...     [
...         LineString(
...             [Point(-4, -3), Point(-2, -3), Point(-4, -2), Point(-2, -1)]
...         )
...     ]
... )
>>> MultipleCrosscutValidator.validation_method(geom, trace_candidates)
True
```

class fractopo.tval.trace_validators.**SharpCornerValidator**

Bases: *BaseValidator*

Find sharp cornered traces.

ERROR = 'SHARP TURNS'

static validation_method(*geom*, *sharp_avg_threshold*, *sharp_prev_seg_threshold*, ***_*)

Validate for sharp cornered traces.

Return type
bool

class fractopo.tval.trace_validators.**SimpleGeometryValidator**

Bases: *BaseValidator*

Use shapely is_simple and is_ring attributes to validate LineString.

Checks that trace does not cut itself.

ERROR = 'CUTS ITSELF'

static validation_method(*geom*, ***_*)

Validate for self-intersections.

Return type
bool

class fractopo.tval.trace_validators.**StackedTracesValidator**

Bases: *BaseValidator*

Find stacked traces and small triangle intersections.

ERROR = 'STACKED TRACES'

static validation_method(*geom*, *trace_candidates*, *snap_threshold*, *snap_threshold_error_multiplier*, *overlap_detection_multiplier*, *triangle_error_snap_multiplier*, ***_*)

Validate for stacked traces and small triangle intersections.

```
>>> geom = LineString([(0, 0), (0, 1)])
>>> trace_candidates = gpd.GeoSeries([LineString([(0, -1), (0, 2)])])
>>> snap_threshold = 0.01
>>> snap_threshold_error_multiplier = 1.1
>>> overlap_detection_multiplier = 50
>>> triangle_error_snap_multiplier = 10
>>> StackedTracesValidator.validation_method(
...     geom,
...     trace_candidates,
...     snap_threshold,
...     snap_threshold_error_multiplier,
...     overlap_detection_multiplier,
...     triangle_error_snap_multiplier,
:rtype: :py:class:`bool`
```

...) False

```
>>> geom = LineString([(10, 0), (10, 1)])
>>> trace_candidates = gpd.GeoSeries([LineString([(0, -1), (0, 2)])])
>>> snap_threshold = 0.01
>>> snap_threshold_error_multiplier = 1.1
>>> overlap_detection_multiplier = 50
```

(continues on next page)

(continued from previous page)

```
>>> triangle_error_snap_multiplier = 10
>>> StackedTracesValidator.validation_method(
...     geom,
...     trace_candidates,
...     snap_threshold,
...     snap_threshold_error_multiplier,
...     overlap_detection_multiplier,
...     triangle_error_snap_multiplier,
... )
True
```

class fractopo.tval.trace_validators.**TargetAreaSnapValidator**

Bases: *BaseValidator*

Validator for traces that underlap the target area.

ERROR = 'TRACE UNDERLAPS TARGET AREA'

static is_candidate_underlapping(*endpoint*, *geom*, *area_polygon*, *snap_threshold*)

Determine if endpoint is candidate for Underlapping error.

Return type
bool

static simple_underlapping_checks(*endpoint*, *geom*, *area_polygon*, *snap_threshold*)

Perform simple underlapping checks.

Return type
Optional[bool]

static validation_method(*geom*, *area*, *snap_threshold*, *snap_threshold_error_multiplier*, *area_edge_snap_multiplier*, **_)

Validate for trace underlaps.

```
>>> geom = LineString([(0, 0), (0, 1)])
>>> area = gpd.GeoDataFrame(
...     geometry=[Polygon([(0, 0), (0, 1), (1, 1), (1, 0)])]
... )
>>> snap_threshold = 0.01
>>> snap_threshold_error_multiplier = 1.1
>>> area_edge_snap_multiplier = 1.0
>>> TargetAreaSnapValidator.validation_method(
...     geom,
...     area,
...     snap_threshold,
...     snap_threshold_error_multiplier,
...     area_edge_snap_multiplier,
:rtype: :py:class:`bool`
```

...) True

```
>>> geom = LineString([(0.5, 0.5), (0.5, 0.98)])
>>> area = gpd.GeoDataFrame(
...     geometry=[Polygon([(0, 0), (0, 1), (1, 1), (1, 0)])]
... )
```

(continues on next page)

(continued from previous page)

```
>>> snap_threshold = 0.01
>>> snap_threshold_error_multiplier = 1.1
>>> area_edge_snap_multiplier = 10
>>> TargetAreaSnapValidator.validation_method(
...     geom,
...     area,
...     snap_threshold,
...     snap_threshold_error_multiplier,
...     area_edge_snap_multiplier,
... )
False
```

class fractopo.tval.trace_validators.**UnderlappingSnapValidator**

Bases: *BaseValidator*

Find snapping errors of underlapping traces.

Uses a multiple of the given snap_threshold.

ERROR = 'UNDERLAPPING SNAP'

classmethod validation_method(*geom*, *trace_candidates*, *snap_threshold*, *snap_threshold_error_multiplier*, ***_*)

Validate for UnderlappingSnapValidator errors.

```
>>> snap_threshold = 0.01
>>> snap_threshold_error_multiplier = 1.1
>>> geom = LineString(
...     [
...         (0, 0),
...         (0, 1 + snap_threshold * snap_threshold_error_multiplier * 0.99),
...     ]
... )
>>> trace_candidates = gpd.GeoSeries([LineString([(-1, 1), (1, 1)])])
>>> UnderlappingSnapValidator.validation_method(
...     geom, trace_candidates, snap_threshold, snap_threshold_error_multiplier
:rtype: :py:class:`bool`
```

...) False

```
>>> snap_threshold = 0.01
>>> snap_threshold_error_multiplier = 1.1
>>> geom = LineString([(0, 0), (0, 1)])
>>> trace_candidates = gpd.GeoSeries([LineString([(-1, 1), (1, 1)])])
>>> UnderlappingSnapValidator.validation_method(
...     geom, trace_candidates, snap_threshold, snap_threshold_error_multiplier
... )
True
```

class fractopo.tval.trace_validators.**VNodeValidator**

Bases: *BaseValidator*

Finds V-nodes within trace data.

ERROR = 'V NODE'

static determine_v_nodes(*endpoint_nodes*, *snap_threshold*, *snap_threshold_error_multiplier*)

Determine V-node errors.

```
>>> endpoint_nodes = [
...     (Point(0, 0), Point(1, 1)),
...     (Point(1, 1),),
... ]
>>> snap_threshold = 0.01
>>> snap_threshold_error_multiplier = 1.1
>>> VNodeValidator.determine_v_nodes(
...     endpoint_nodes, snap_threshold, snap_threshold_error_multiplier
:rtype: :py:class:`~typing.Set`\[:py:class:`int`]
```

...) {0, 1}

static validation_method(*idx*, *vnodes*, **_)

Validate for V-nodes.

Return type
bool

```
>>> VNodeValidator.validation_method(idx=1, vnodes=set([1, 2]))
False
```

```
>>> VNodeValidator.validation_method(idx=5, vnodes=set([1, 2]))
True
```

Module contents

Package with trace validation utility.

Submodules

fractopo.branches_and_nodes module

Functions for extracting branches and nodes from trace maps.

branches_and_nodes is the main entrypoint.

fractopo.branches_and_nodes.**additional_snapping_func**(*trace*, *idx*, *additional_snapping*)

Insert points into LineStrings to make sure trace abutting trace.

E.g.

```
>>> trace = LineString([(0, 0), (1, 0), (2, 0), (3, 0)])
>>> idx = 0
>>> point = Point(2.25, 0.1)
>>> additional_snapping = [
...     (0, point),
... ]
:rtype: :py:class:`~shapely.geometry.linestring.LineString`
```

```
>>> additional_snapping_func(trace, idx, additional_snapping).wkt
'LINESTRING (0 0, 1 0, 2 0, 2.25 0.1, 3 0)'
```

When idx doesn't match -> no additional snapping

```
>>> trace = LineString([(0, 0), (1, 0), (2, 0), (3, 0)])
>>> idx = 1
>>> point = Point(2.25, 0.1)
>>> additional_snapping = [
...     (0, point),
... ]
>>> additional_snapping_func(trace, idx, additional_snapping).wkt
'LINESTRING (0 0, 1 0, 2 0, 3 0)'
```

fractopo.branches_and_nodes.**angle_to_point**(*point*, *nearest_point*, *comparison_point*)

Calculate the angle between two vectors.

Vectors are made from the given points: Both vectors have the same first point, nearest_point, and second point is either point or comparison_point.

Returns angle in degrees.

E.g.

```
>>> point = Point(1, 1)
>>> nearest_point = Point(0, 0)
>>> comparison_point = Point(-1, 1)
:rtype: :py:class:`float`
```

```
>>> angle_to_point(point, nearest_point, comparison_point)
90.0
```

```
>>> point = Point(1, 1)
>>> nearest_point = Point(0, 0)
>>> comparison_point = Point(-1, 2)
>>> angle_to_point(point, nearest_point, comparison_point)
71.56505117707799
```

fractopo.branches_and_nodes.**determine_branch_identity**(*number_of_i_nodes*, *number_of_xy_nodes*, *number_of_e_nodes*)

Determine the identity of a branch.

Is based on the amount of I-, XY- and E-nodes and returns it as a string.

E.g.

Return type
str

```
>>> determine_branch_identity(2, 0, 0)
'I - I'
```

```
>>> determine_branch_identity(1, 1, 0)
'C - I'
```

```
>>> determine_branch_identity(1, 0, 1)
'I - E'
```

fractopo.branches_and_nodes.**determine_insert_approach**(*nearest_point_idx*, *trace_point_dists*, *snap_threshold*, *point*, *nearest_point*)

Determine if to insert or replace point.

fractopo.branches_and_nodes.**filter_non_unique_traces**(*traces*, *snap_threshold*)

Filter out traces that are not unique.

Return type
GeoSeries

fractopo.branches_and_nodes.**get_branch_identities**(*branches*, *nodes*, *node_identities*, *snap_threshold*)

Determine the types of branches for a GeoSeries of branches.

i.e. C-C, C-I or I-I, + (C-E, E-E, I-E)

```
>>> branches = gpd.GeoSeries(
...     [
...         LineString([(1, 1), (2, 2)]),
...         LineString([(2, 2), (3, 3)]),
...         LineString([(3, 0), (2, 2)]),
...         LineString([(2, 2), (-2, 5)]),
...     ]
... )
>>> nodes = gpd.GeoSeries(
...     [
...         Point(2, 2),
...         Point(1, 1),
...         Point(3, 3),
...         Point(3, 0),
...         Point(-2, 5),
...     ]
... )
>>> node_identities = ["X", "I", "I", "I", "E"]
>>> snap_threshold = 0.001
:rtype: :py:class:`~typing.List`\[:py:class:`str`]
```

```
>>> get_branch_identities(branches, nodes, node_identities, snap_threshold)
['C - I', 'C - I', 'C - I', 'C - E']
```

fractopo.branches_and_nodes.**insert_point_to_linestring**(*trace*, *point*, *snap_threshold*)

Insert/modify point to trace LineString.

The point location is determined to fit into the LineString without changing the geometrical order of LineString vertices (which only makes sense if LineString is sublinear.)

TODO: Does not work for 2.5D geometries (Z-coordinates). Z-coordinates will be lost.

E.g.

```
>>> trace = LineString([(0, 0), (1, 0), (2, 0), (3, 0)])
>>> point = Point(1.25, 0.1)
:rtype: :py:class:`~shapely.geometry.linestring.LineString`
```

```
>>> insert_point_to_linestring(trace, point, 0.01).wkt
'LINESTRING (0 0, 1 0, 1.25 0.1, 2 0, 3 0)'
```

```
>>> trace = LineString([(0, 0), (1, 0), (2, 0), (3, 0)])
>>> point = Point(2.25, 0.1)
>>> insert_point_to_linestring(trace, point, 0.01).wkt
'LINESTRING (0 0, 1 0, 2 0, 2.25 0.1, 3 0)'
```

fractopo.branches_and_nodes.**is_endpoint_close_to_boundary**(*endpoint*, *areas*, *snap_threshold*)

Check if endpoint is within snap_threshold of areas boundaries.

Return type
bool

fractopo.branches_and_nodes.**node_identities_from_branches**(*branches*, *areas*, *snap_threshold*)

Resolve node identities from branch data.

```
>>> branches_list = [
...     LineString([(0, 0), (1, 1)]),
...     LineString([(2, 2), (1, 1)]),
...     LineString([(2, 0), (1, 1)]),
... ]
>>> area_polygon = Polygon([(-5, -5), (-5, 5), (5, 5), (5, -5)])
>>> branches = gpd.GeoSeries(branches_list)
>>> areas = gpd.GeoSeries([area_polygon])
>>> snap_threshold = 0.001
>>> nodes, identities = node_identities_from_branches(
...     branches, areas, snap_threshold
... )
>>> [node.wkt for node in nodes]
['POINT (0 0)', 'POINT (1 1)', 'POINT (2 2)', 'POINT (2 0)']
:rtype: :py:data:`~typing.Tuple`\[:py:class:`~typing.List`\[:py:class:`~shapely.
↪geometry.point.Point`], :py:class:`~typing.List`\[:py:class:`str`]]
```

```
>>> identities
['I', 'Y', 'I', 'I']
```

fractopo.branches_and_nodes.**node_identity**(*endpoint*, *idx*, *areas*, *endpoints_geoseries*, *endpoints_spatial_index*, *snap_threshold*)

Determine node identity of endpoint.

Return type
str

fractopo.branches_and_nodes.**part_unary_union**(*split_traces*, *snap_threshold*, *size_threshold*, *div*)

Conduct safer_unary_union in parts.

fractopo.branches_and_nodes.**report_snapping_loop**(*loops*, *allowed_loops*)

Report snapping looping.

fractopo.branches_and_nodes.**resolve_trace_candidates**(*trace*, *idx*, *traces_spatial_index*, *traces*, *snap_threshold*)

Resolve PyGEOSSTRTreeIndex intersection to actual intersection candidates.

Return type
List[LineString]

fractopo.branches_and_nodes.**safer_unary_union**(*traces_geosrs*, *snap_threshold*, *size_threshold*)

Perform unary union to transform traces to branch segments.

unary_union is not completely stable with large datasets but problem can be alleviated by dividing analysis to parts.

TODO: Usage is deprecated as unary_union seems to give consistent results.

Return type
MultiLineString

fractopo.branches_and_nodes.**simple_snap**(*trace*, *trace_candidates*, *snap_threshold*)

Modify conditionally trace to snap to any of trace_candidates.

E.g.

```
>>> trace = LineString([(0, 0), (1, 0), (2, 0), (3, 0)])
>>> trace_candidates = gpd.GeoSeries(
...     [LineString([(3.0001, -3), (3.0001, 0), (3, 3)])]
... )
>>> snap_threshold = 0.001
>>> snapped = simple_snap(trace, trace_candidates, snap_threshold)
:rtype: :py:data:`~typing.Tuple`\[:py:class:`~shapely.geometry.linestring.
↪LineString`, :py:class:`bool`]
```

```
>>> snapped[0].wkt, snapped[1]
('LINESTRING (0 0, 1 0, 2 0, 3.0001 0)', True)
```

Do not snap overlapping.

```
>>> trace = LineString([(0, 0), (1, 0), (2, 0), (3.0002, 0)])
>>> trace_candidates = gpd.GeoSeries(
...     [LineString([(3.0001, -3), (3.0001, 0), (3, 3)])]
... )
>>> snap_threshold = 0.001
>>> snapped = simple_snap(trace, trace_candidates, snap_threshold)
>>> snapped[0].wkt, snapped[1]
('LINESTRING (0 0, 1 0, 2 0, 3.0002 0)', False)
```

fractopo.branches_and_nodes.**snap_others_to_trace**(*idx*, *trace*, *snap_threshold*, *traces*, *traces_spatial_index*, *areas*, *final_allowed_loop=False*)

Determine whether and how to snap *trace* to *traces*.

E.g.

Trace gets new coordinates to snap other traces to it:

```
>>> idx = 0
>>> trace = LineString([(0, 0), (1, 0), (2, 0), (3, 0)])
>>> snap_threshold = 0.001
>>> traces = [trace, LineString([(1.5, 3), (1.5, 0.00001)])]
>>> traces_spatial_index = pygeos_spatial_index(gpd.GeoSeries(traces))
>>> areas = None
```

(continues on next page)

(continued from previous page)

```
>>> snapped = snap_others_to_trace(
...     idx, trace, snap_threshold, traces, traces_spatial_index, areas
... )
:rtype: :py:data:`~typing.Tuple`\[:py:class:`~shapely.geometry.linestring.
↪LineString`, :py:class:`bool`]
```

```
>>> snapped[0].wkt, snapped[1]
('LINESTRING (0 0, 1 0, 1.5 1e-05, 2 0, 3 0)', True)
```

Trace itself is not snapped by snap_others_to_trace:

```
>>> idx = 0
>>> trace = LineString([(0, 0), (1, 0), (2, 0), (3, 0)])
>>> snap_threshold = 0.001
>>> traces = [trace, LineString([(3.0001, -3), (3.0001, 0), (3, 3)])]
>>> traces_spatial_index = pygeos_spatial_index(gpd.GeoSeries(traces))
>>> areas = None
>>> snapped = snap_others_to_trace(
...     idx, trace, snap_threshold, traces, traces_spatial_index, areas
... )
>>> snapped[0].wkt, snapped[1]
('LINESTRING (0 0, 1 0, 2 0, 3 0)', False)
```

fractopo.branches_and_nodes.**snap_trace_simple**(*idx*, *trace*, *snap_threshold*, *traces*, *traces_spatial_index*, *final_allowed_loop=False*)

Determine whether and how to perform simple snap.

Return type
Tuple[LineString, bool]

fractopo.branches_and_nodes.**snap_trace_to_another**(*trace_endpoints*, *another*, *snap_threshold*)

Add point to another trace to snap trace to end at another trace.

I.e. modifies and returns *another*

Return type
Tuple[LineString, bool]

fractopo.branches_and_nodes.**snap_traces**(*traces*, *snap_threshold*, *areas=None*, *final_allowed_loop=False*)

Snap traces to end exactly at other traces.

Return type
Tuple[List[LineString], bool]

fractopo.cli module

Command-line integration of fractopo with click.

class fractopo.cli.**LogLevel**(*value*)

Bases: Enum

Enums for log levels.

CRITICAL = 'CRITICAL'

DEBUG = 'DEBUG'

ERROR = 'ERROR'

INFO = 'INFO'

WARNING = 'WARNING'

fractopo.cli.**default_network_output_paths**(*network_name*, *general_output*, *branches_output*, *nodes_output*, *parameters_output*)

Determine default network output paths.

fractopo.cli.**describe_results**(*validated*, *error_column*, *console=<console width=80 None>*)

Describe validation results to stdout.

fractopo.cli.**fractopo_callback**(*log_level=<typer.models.OptionInfo object>*)

Use fractopo command-line utilities.

fractopo.cli.**get_click_path_args**(*exists=True*, ***kwargs*)

Get basic click path args.

fractopo.cli.**info**()

Print out information about fractopo installation and python environment.

fractopo.cli.**make_output_dir**(*base_path*)

Make timestamped output dir.

Return type
Path

fractopo.cli.**network**(*trace_file=<typer.models.ArgumentInfo object>*, *area_file=<typer.models.ArgumentInfo object>*, *snap_threshold=<typer.models.OptionInfo object>*, *determine_branches_nodes=<typer.models.OptionInfo object>*, *name=<typer.models.OptionInfo object>*, *circular_target_area=<typer.models.OptionInfo object>*, *truncate_traces=<typer.models.OptionInfo object>*, *censoring_area=<typer.models.OptionInfo object>*, *branches_output=<typer.models.OptionInfo object>*, *nodes_output=<typer.models.OptionInfo object>*, *general_output=<typer.models.OptionInfo object>*, *parameters_output=<typer.models.OptionInfo object>*)

Analyze the geometry and topology of trace network.

fractopo.cli.**rich_table_from_parameters**(*parameters*)

Generate `rich` `Table` from network parameters.

Return type
Table

fractopo.cli.**tracevalidate**(*trace_file=<typer.models.ArgumentInfo object>*, *area_file=<typer.models.ArgumentInfo object>*, *allow_fix=<typer.models.OptionInfo object>*, *summary=<typer.models.OptionInfo object>*, *snap_threshold=<typer.models.OptionInfo object>*, *output=<typer.models.OptionInfo object>*, *only_area_validation=<typer.models.OptionInfo object>*, *allow_empty_area=<typer.models.OptionInfo object>*)

Validate trace data delineated by target area data.

If allow_fix is True, some automatic fixing will be done to e.g. convert MultiLineStrings to LineStrings.

fractopo.fractopo_utils module

Miscellaneous utilities and scripts of fractopo.

class fractopo.fractopo_utils.**LineMerge**

Bases: object

Merge lines conditionally.

static conditional_linemerge(*first*, *second*, *tolerance*, *buffer_value*)

Conditionally merge two LineStrings (first and second).

Merge occurs if 1) their endpoints are within buffer_value of each other and 2) their total orientations are within tolerance (degrees) of each other.

Merges by joining their coordinates. The endpoint (that is within buffer_value of endpoint of first) of the second LineString is trimmed from the resulting coordinates.

E.g. with merging:

```
>>> first = LineString([(0, 0), (0, 2)])
>>> second = LineString([(0, 2.001), (0, 4)])
>>> tolerance = 5
>>> buffer_value = 0.01
:rtype: :py:data:`~typing.Optional`\[:py:class:`~shapely.geometry.linestring.
↪LineString`]
```

```
>>> LineMerge.conditional_linemerge(first, second, tolerance, buffer_value).wkt
'LINESTRING (0 0, 0 2, 0 4)'
```

Without merging:

```
>>> first = LineString([(0, 0), (0, 2)])
>>> second = LineString([(0, 2.1), (0, 4)])
>>> tolerance = 5
>>> buffer_value = 0.01
>>> LineMerge.conditional_linemerge(
...     first, second, tolerance, buffer_value
... ) is None
True
```

static conditional_linemerge_collection(*traces*, *tolerance*, *buffer_value*)

Conditionally linemerge within a collection of LineStrings.

Returns the linemerged traces and the idxs of traces that were linemerged.

E.g.

```
>>> first = LineString([(0, 0), (0, 2)])
>>> second = LineString([(0, 2.001), (0, 4)])
>>> traces = gpd.GeoSeries([first, second])
```

(continues on next page)

(continued from previous page)

```
>>> tolerance = 5
>>> buffer_value = 0.01
>>> new_traces, idx = LineMerge.conditional_linemerge_collection(
...     traces, tolerance, buffer_value
... )
:rtype: :py:data:`~typing.Tuple`\[:py:class:`~typing.List`\[:py:class:`~shapely.
↪geometry.linestring.LineString`], :py:class:`~typing.List`\[:py:class:`int`]]
```

```
>>> [trace.wkt for trace in new_traces], idx
(['LINESTRING (0 0, 0 2, 0 4)'], [0, 1])
```

static integrate_replacements(*traces*, *new_traces*, *modified_idx*)

Add linemerged and remove the parts that were linemerged.

E.g.

```
>>> first = LineString([(0, 0), (0, 2)])
>>> second = LineString([(0, 2.001), (0, 4)])
>>> traces = gpd.GeoDataFrame(geometry=[first, second])
>>> new_traces = [LineString([(0, 0), (0, 2), (0, 4)])]
>>> modified_idx = [0, 1]
>>> LineMerge.integrate_replacements(traces, new_traces, modified_idx)
:rtype: :py:class:`~geopandas.geodataframe.GeoDataFrame`
```

geometry

0 LINESTRING (0.00000 0.00000, 0.00000 2.00000, …

static run_loop(*traces*, *tolerance*, *buffer_value*)

Run multiple conditional linemerge iterations for GeoDataFrame.

This is the main entrypoint.

GeoDataFrame should contain LineStrings.

E.g.

```
>>> first = LineString([(0, 0), (0, 2)])
>>> second = LineString([(0, 2.001), (0, 4)])
>>> traces = gpd.GeoDataFrame(geometry=[first, second])
>>> tolerance = 5
>>> buffer_value = 0.01
>>> LineMerge.run_loop(traces, tolerance, buffer_value)
:rtype: :py:class:`~geopandas.geodataframe.GeoDataFrame`
```

geometry

0 LINESTRING (0.00000 0.00000, 0.00000 2.00000, …

fractopo.fractopo_utils.**remove_identical_sindex**(*geosrs*, *snap_threshold*)

Remove stacked nodes by using a search buffer the size of snap_threshold.

Return type
GeoSeries

fractopo.general module

Contains general calculation and plotting tools.

class fractopo.general.**Aggregator**(*value*)

Bases: Enum

Define how to aggregate during subsample aggragation.

MEAN(*weights*)

Aggregate by calculating mean.

Return type
Union[float, int]

SUM(**_)

Aggregate by calculating sum.

Return type
Union[float, int]

class fractopo.general.**Col**(*value*)

Bases: Enum

GeoDataFrame column names for attributes.

AZIMUTH = 'azimuth'

AZIMUTH_SET = 'azimuth_set'

LENGTH = 'length'

LENGTH_NON_WEIGHTED = 'length non-weighted'

LENGTH_SET = 'length_set'

LENGTH_WEIGHTS = 'boundary_weight'

class fractopo.general.**Param**(*value*)

Bases: Enum

Column names for geometric and topological parameters.

The ParamInfo instances contain additional metadata.

AREA = ParamInfo(name='Area', plot_as_log=False, unit='m^2', needs_topology=False, aggregator=<function sum_aggregation>)

AREAL_FREQUENCY_B20 = ParamInfo(name='Areal Frequency B20', plot_as_log=True, unit='$\\frac{1}{m^2}$', needs_topology=True, aggregator=<function mean_aggregation>)

AREAL_FREQUENCY_P20 = ParamInfo(name='Areal Frequency P20', plot_as_log=True, unit='$\\frac{1}{m^2}$', needs_topology=True, aggregator=<function mean_aggregation>)

BRANCH_MAX_LENGTH = ParamInfo(name='Branch Max Length', plot_as_log=True, unit='m', needs_topology=True, aggregator=<function mean_aggregation>)

```
BRANCH_MEAN_LENGTH = ParamInfo(name='Branch Mean Length', plot_as_log=True,
unit='$m$', needs_topology=True, aggregator=<function mean_aggregation>)

BRANCH_MIN_LENGTH = ParamInfo(name='Branch Min Length', plot_as_log=True,
unit='$m$', needs_topology=True, aggregator=<function mean_aggregation>)

CIRCLE_COUNT = ParamInfo(name='Circle Count', plot_as_log=False, unit='-',
needs_topology=False, aggregator=<function sum_aggregation>)

CONNECTIONS_PER_BRANCH = ParamInfo(name='Connections per Branch', plot_as_log=False,
unit='$\\frac{1}{n}$', needs_topology=True, aggregator=<function mean_aggregation>)

CONNECTIONS_PER_TRACE = ParamInfo(name='Connections per Trace', plot_as_log=False,
unit='$\\frac{1}{n}$', needs_topology=True, aggregator=<function mean_aggregation>)

CONNECTION_FREQUENCY = ParamInfo(name='Connection Frequency', plot_as_log=False,
unit='$\\frac{1}{m^2}$', needs_topology=True, aggregator=<function
mean_aggregation>)

DIMENSIONLESS_INTENSITY_B22 = ParamInfo(name='Dimensionless Intensity B22',
plot_as_log=False, unit='-', needs_topology=True, aggregator=<function
mean_aggregation>)

DIMENSIONLESS_INTENSITY_P22 = ParamInfo(name='Dimensionless Intensity P22',
plot_as_log=False, unit='-', needs_topology=False, aggregator=<function
mean_aggregation>)

FRACTURE_DENSITY_MAULDON = ParamInfo(name='Fracture Density (Mauldon)',
plot_as_log=True, unit='$\\frac{1}{m^2}$', needs_topology=True, aggregator=<function
mean_aggregation>)

FRACTURE_INTENSITY_B21 = ParamInfo(name='Fracture Intensity B21', plot_as_log=True,
unit='$\\frac{m}{m^2}$', needs_topology=False, aggregator=<function
mean_aggregation>)

FRACTURE_INTENSITY_MAULDON = ParamInfo(name='Fracture Intensity (Mauldon)',
plot_as_log=True, unit='$\\frac{m}{m^2}$', needs_topology=True, aggregator=<function
mean_aggregation>)

FRACTURE_INTENSITY_P21 = ParamInfo(name='Fracture Intensity P21', plot_as_log=True,
unit='$\\frac{m}{m^2}$', needs_topology=False, aggregator=<function
mean_aggregation>)

NUMBER_OF_BRANCHES = ParamInfo(name='Number of Branches', plot_as_log=False,
unit='-', needs_topology=True, aggregator=<function sum_aggregation>)

NUMBER_OF_BRANCHES_TRUE = ParamInfo(name='Number of Branches (Real)',
plot_as_log=False, unit='-', needs_topology=True, aggregator=<function
sum_aggregation>)

NUMBER_OF_TRACES = ParamInfo(name='Number of Traces', plot_as_log=False, unit='-',
needs_topology=True, aggregator=<function sum_aggregation>)

NUMBER_OF_TRACES_TRUE = ParamInfo(name='Number of Traces (Real)', plot_as_log=False,
unit='-', needs_topology=True, aggregator=<function sum_aggregation>)
```

TRACE_MAX_LENGTH = ParamInfo(name='Trace Max Length', plot_as_log=True, unit='m', needs_topology=False, aggregator=<function mean_aggregation>)

TRACE_MEAN_LENGTH = ParamInfo(name='Trace Mean Length', plot_as_log=True, unit='m', needs_topology=False, aggregator=<function mean_aggregation>)

TRACE_MEAN_LENGTH_MAULDON = ParamInfo(name='Trace Mean Length (Mauldon)', plot_as_log=True, unit='m', needs_topology=True, aggregator=<function mean_aggregation>)

TRACE_MIN_LENGTH = ParamInfo(name='Trace Min Length', plot_as_log=True, unit='m', needs_topology=False, aggregator=<function mean_aggregation>)

class fractopo.general.**ParamInfo**(*name*, *plot_as_log*, *unit*, *needs_topology*, *aggregator*)

Bases: object

Parameter with name and metadata.

aggregator: *Aggregator*

name: str

needs_topology: bool

plot_as_log: bool

unit: str

class fractopo.general.**ProcessResult**(*identifier*, *result*, *error*)

Bases: object

Dataclass for multiprocessing result parsing.

error: bool

identifier: str

result: Any

fractopo.general.**assign_branch_and_node_colors**(*feature_type*)

Determine color for each branch and node type.

Defaults to red.

Return type
str

```
>>> assign_branch_and_node_colors("C-C")
'red'
```

fractopo.general.**avg_calc**(*data*)

Calculate average for radial data.

TODO: Should take length into calculations…….. not real average atm

fractopo.general.**azimu_half**(*degrees*)

Transform azimuth from 180-360 range to range 0-180.

Parameters
degrees (float) – Degrees in range 0 - 360

Return type
float

Returns
Degrees in range 0 - 180

fractopo.general.**azimuth_to_unit_vector**(*azimuth*)

Convert azimuth to unit vector.

Return type
ndarray

fractopo.general.**bool_arrays_sum**(*arr_1*, *arr_2*)

Calculate integer sum of two arrays.

Resulting array consists only of integers 0, 1 and 2.

```
>>> arr_1 = np.array([True, False, False])
>>> arr_2 = np.array([True, True, False])
:rtype: :py:class:`~numpy.ndarray`
```

```
>>> bool_arrays_sum(arr_1, arr_2)
array([2, 1, 0])
```

```
>>> arr_1 = np.array([True, True])
>>> arr_2 = np.array([True, True])
>>> bool_arrays_sum(arr_1, arr_2)
array([2, 2])
```

fractopo.general.**bounding_polygon**(*geoseries*)

Create bounding polygon around GeoSeries.

The geoseries geometries will always be completely enveloped by the polygon. The geometries will not intersect the polygon boundary.

```
>>> geom = LineString([(1, 0), (1, 1), (-1, -1)])
>>> geoseries = gpd.GeoSeries([geom])
>>> poly = bounding_polygon(geoseries)
>>> poly.wkt
'POLYGON ((2 -2, 2 2, -2 2, -2 -2, 2 -2))'
>>> geoseries.intersects(poly.boundary)
:rtype: :py:class:`~shapely.geometry.polygon.Polygon`
```

0 False dtype: bool

fractopo.general.**calc_circle_area**(*radius*)

Calculate area of circle.

Return type
float

```
>>> calc_circle_area(1.78)
9.953822163633902
```

fractopo.general.**calc_circle_radius**(*area*)

Calculate radius from area.

Return type
float

```
>>> calc_circle_radius(10.0)
1.7841241161527712
```

fractopo.general.**calc_strike**(*dip_direction*)

Calculate strike from dip direction. Right-handed rule.

E.g.:

```
>>> calc_strike(50.0)
320.0
```

```
>>> calc_strike(180.0)
90.0
```

Parameters
dip_direction (float) – The direction of dip.

Return type
float

Returns
Converted strike.

fractopo.general.**check_for_wrong_geometries**(*traces*, *area*)

Check that traces are line geometries and area contains area geometries.

fractopo.general.**check_for_z_coordinates**(*geodata*)

Check if geopandas data contains Z-coordinates.

Return type
bool

fractopo.general.**compare_unit_vector_orientation**(*vec_1*, *vec_2*, *threshold_angle*)

If vec_1 and vec_2 are too different in orientation, will return False.

Return type
bool

fractopo.general.**convert_list_columns**(*gdf*, *allow=True*)

Convert list type columns to string.

Return type
GeoDataFrame

fractopo.general.**create_unit_vector**(*start_point*, *end_point*)

Create numpy unit vector from two shapely Points.

Parameters

- **start_point** (Point) – The start point.
- **end_point** (Point) – The end point.

Return type
ndarray

Returns
The unit vector that points from `start_point` to `end_point`.

`fractopo.general.`**`define_length_set`**(*length*, *set_df*)

Define sets based on the length of the traces or branches.

Return type
`str`

`fractopo.general.`**`determine_azimuth`**(*line*, *halved*)

Calculate azimuth of given line.

If halved -> return is in range [0, 180] Else -> [0, 360]

e.g.: Accepts LineString

```
>>> determine_azimuth(LineString([(0, 0), (1, 1)]), True)
45.0
```

```
>>> determine_azimuth(LineString([(0, 0), (0, 1)]), True)
0.0
```

```
>>> determine_azimuth(LineString([(0, 0), (-1, -1)]), False)
225.0
```

```
>>> determine_azimuth(LineString([(0, 0), (-1, -1)]), True)
45.0
```

Parameters

- **`line`** (`LineString`) – The line of which azimuth is determined.
- **`halved`** (`bool`) – Whether to return result in range [0, 180] (halved=True) or [0, 360] (halved=False).

Return type
`float`

Returns
The determined azimuth.

`fractopo.general.`**`determine_boundary_intersecting_lines`**(*line_gdf*, *area_gdf*, *snap_threshold*)

Determine lines that intersect any target area boundary.

`fractopo.general.`**`determine_regression_azimuth`**(*line*)

Determine azimuth of line LineString with linear regression.

A scikit-learn LinearRegression is fitted to the x, y coordinates of the given and the azimuth of the fitted linear line is returned.

The azimuth is returned in range [0, 180].

E.g.

```
>>> line = LineString([(0, 0), (1, 1), (2, 2), (3, 3)])
>>> determine_regression_azimuth(line)
45.0
```

```
>>> line = LineString([(-1, -5), (3, 3)])
>>> round(determine_regression_azimuth(line), 3)
26.565
```

```
>>> line = LineString([(0, 0), (0, 3)])
>>> determine_regression_azimuth(line)
0.0
```

Parameters
line (LineString) – The line of which azimuth is determined.

Return type
float

Returns
The determined azimuth in range [0, 180].

Raises
ValueError – When LinearRegression returns unexpected coefficients.

fractopo.general.**determine_set**(*value*, *value_ranges*, *set_names*, *loop_around*)

Determine which named value range, if any, value is within.

loop_around defines behavior expected for radial data i.e. when value range can loop back around e.g. [160, 50]

E.g.

```
>>> determine_set(10.0, [(0, 20), (30, 160)], ["0-20", "30-160"], False)
'0-20'
```

Example with

```
>>> determine_set(50.0, [(0, 20), (160, 60)], ["0-20", "160-60"], True)
'160-60'
```

Parameters

- **value** (float) – Value to determine set of.
- **value_ranges** (Tuple[Tuple[float, float], ...]) – Ranges of each set.
- **set_names** (Tuple[str, ...]) – Names of each set.
- **loop_around** (bool) – Whether the sets loop around. This is the case for radial data such as azimuths but not the case for length data.

Return type
str

Returns
Set string in which value belongs.

Raises
ValueError – When set value ranges overlap.

fractopo.general.**determine_valid_intersection_points**(*intersection_geoms*)

Filter intersection points between trace candidates and geom.

Only allows Point geometries as intersections. LineString intersections would be possible if geometries are stacked.

Parameters
intersection_geoms (GeoSeries) – GeoSeries of intersection (Point) geometries.

Return type
List[Point]

Returns
The valid intersections (Points).

fractopo.general.**determine_valid_intersection_points_no_vnode**(*trace_candidates*, *geom*)

Filter intersection points between trace candidates and geom with no vnodes.

V-node intersections are validated by looking at the endpoints. If V-nodes were kept as intersection points the VNodeValidator could not find V-node errors.

Return type
List[Point]

fractopo.general.**dissolve_multi_part_traces**(*traces*)

Dissolve MultiLineStrings in GeoDataFrame or GeoSeries.

Copies all attribute data of rows with MultiLineStrings to new LineStrings.

Return type
Union[GeoDataFrame, GeoSeries]

fractopo.general.**efficient_clip**(*traces*, *areas*)

Perform efficient clip of LineString geometries with a Polygon.

Parameters

- **traces** (Union[GeoSeries, GeoDataFrame]) – Trace data.
- **areas** (Union[GeoSeries, GeoDataFrame]) – Area data.

Return type
GeoDataFrame

Returns
Traces clipped with the area data.

fractopo.general.**extend_bounds**(*min_x*, *min_y*, *max_x*, *max_y*, *extend_amount*)

Extend bounds by addition and reduction.

Return type
Tuple[float, float, float, float]

```
>>> extend_bounds(0, 0, 10, 10, 10)
(-10, -10, 20, 20)
```

fractopo.general.**fallback_aggregation**(*values*)

Fallback aggregation where values are simply joined into a string.

Return type
str

fractopo.general.**flatten_tuples**(*list_of_tuples*)

Flatten collection of tuples and return index references.

Indexes are from original tuple groupings.

E.g.

```
>>> tuples = [(1, 1, 1), (2, 2, 2, 2), (3,)]
:rtype: :py:data:`~typing.Tuple`\[:py:class:`~typing.List`\[:py:class:`int`], :py:
↪class:`~typing.List`\[:py:data:`~typing.Any`]]
```

```
>>> flatten_tuples(tuples)
([0, 0, 0, 1, 1, 1, 1, 2], [1, 1, 1, 2, 2, 2, 2, 3])
```

fractopo.general.**focus_plot_to_bounds**(*ax*, *total_bounds*)

Focus plot to given bounds.

Return type
Axes

fractopo.general.**geom_bounds**(*geom*)

Get LineString or Polygon bounds.

Return type
Tuple[float, float, float, float]

```
>>> geom_bounds(LineString([(-10, -10), (10, 10)]))
(-10.0, -10.0, 10.0, 10.0)
```

fractopo.general.**get_next_point_in_trace**(*trace*, *point*)

Determine next coordinate point towards middle of LineString from point.

Return type
Point

fractopo.general.**get_trace_coord_points**(*trace*)

Get all coordinate Points of a LineString.

```
>>> trace = LineString([(0, 0), (2, 0), (3, 0)])
>>> coord_points = get_trace_coord_points(trace)
:rtype: :py:class:`~typing.List`\[:py:class:`~shapely.geometry.point.Point`]
```

```
>>> print([p.wkt for p in coord_points])
['POINT (0 0)', 'POINT (2 0)', 'POINT (3 0)']
```

fractopo.general.**get_trace_endpoints**(*trace*)

Return endpoints (shapely.geometry.Point) of a given LineString.

Return type
Tuple[Point, Point]

fractopo.general.**intersection_count_to_boundary_weight**(*intersection_count*)

Get actual weight factor for boundary intersection count.

```
>>> intersection_count_to_boundary_weight(2)
0
```

(continues on next page)

(continued from previous page)

```
>>> intersection_count_to_boundary_weight(0)
1
:rtype: :py:class:`int`
```

```
>>> intersection_count_to_boundary_weight(1)
2
```

fractopo.general.**is_azimuth_close**(*first*, *second*, *tolerance*, *halved=True*)

Determine are azimuths first and second within tolerance.

Takes into account the radial nature of azimuths.

```
>>> is_azimuth_close(0, 179, 15)
True
```

```
>>> is_azimuth_close(166, 179, 15)
True
```

```
>>> is_azimuth_close(20, 179, 15)
False
```

Parameters

- **first** (float) – First azimuth value to compare.
- **second** (float) – Second azimuth value to compare.
- **tolerance** (float) – Tolerance for closeness.
- **halved** (bool) – Are the azimuths azial (i.e. halved=True) or vectors.

fractopo.general.**is_empty_area**(*area*, *traces*)

Check if any traces intersect the area(s) in area GeoDataFrame.

fractopo.general.**is_set**(*value*, *value_range*, *loop_around*)

Determine if value fits within the given value_range.

If the value range has the possibility of looping around loop_around can be set to true.

```
>>> is_set(5, (0, 10), False)
True
```

```
>>> is_set(5, (175, 15), True)
True
```

Parameters

- **value** (Union[float, int]) – Value to determine.
- **Tuple[float,** (*value_range*) – The range of values.
- **loop_around** (bool) – Whether the range loops around. This is the case for radial data such as azimuths but not the case for length data.

Return type

bool

Returns
Is it within range.

fractopo.general.**line_intersection_to_points**(*first*, *second*)

Perform shapely intersection between two LineStrings.

Enforces only Point returns.

Return type
List[Point]

fractopo.general.**match_crs**(*first*, *second*)

Match crs between two geopandas data structures.

```
>>> first = gpd.GeoSeries([Point(1, 1)], crs="EPSG:3067")
>>> second = gpd.GeoSeries([Point(1, 1)])
>>> m_first, m_second = match_crs(first, second)
:rtype: :py:data:`~typing.Tuple`\[:py:data:`~typing.Union`\[:py:class:`~geopandas.
↪geoseries.GeoSeries`, :py:class:`~geopandas.geodataframe.GeoDataFrame`], :py:data:
↪`~typing.Union`\[:py:class:`~geopandas.geoseries.GeoSeries`, :py:class:`~
↪geopandas.geodataframe.GeoDataFrame`]]
```

```
>>> m_first.crs == m_second.crs
True
```

fractopo.general.**mean_aggregation**(*values*, *weights*)

Aggregate by calculating mean.

Return type
Union[float, int]

fractopo.general.**mls_to_ls**(*multilinestrings*)

Flattens a list of multilinestrings to a list of linestrings.

```
>>> multilinestrings = [
...     MultiLineString(
...         [
...             LineString([(1, 1), (2, 2), (3, 3)]),
...             LineString([(1.9999, 2), (-2, 5)]),
...         ]
...     ),
...     MultiLineString(
...         [
...             LineString([(1, 1), (2, 2), (3, 3)]),
...             LineString([(1.9999, 2), (-2, 5)]),
...         ]
...     ),
... ]
>>> result_linestrings = mls_to_ls(multilinestrings)
>>> print([ls.wkt for ls in result_linestrings])
:rtype: :py:class:`~typing.List`\[:py:class:`~shapely.geometry.linestring.
↪LineString`]
```

['LINESTRING (1 1, 2 2, 3 3)', 'LINESTRING (1.9999 2, -2 5)', 'LINESTRING (1 1, 2 2, 3 3)', 'LINESTRING (1.9999 2, -2 5)']

fractopo.general.**multiprocess**(*function_to_call*, *keyword_arguments*, *arguments_identifier=<function <lambda>>*, *repeats=0*)

Process function calls in parallel.

Returns result as a list where the error is appended when execution fails.

Return type
List[*ProcessResult*]

fractopo.general.**numpy_to_python_type**(*value*)

Convert numpy dtype variable to Python type, if possible.

fractopo.general.**point_to_point_unit_vector**(*point*, *other_point*)

Create unit vector from point to other point.

```
>>> point = Point(0, 0)
>>> other_point = Point(1, 1)
:rtype: :py:class:`~numpy.ndarray`
```

```
>>> point_to_point_unit_vector(point, other_point)
array([0.70710678, 0.70710678])
```

fractopo.general.**point_to_xy**(*point*)

Get x and y coordinates of Point.

Return type
Tuple[float, float]

fractopo.general.**prepare_geometry_traces**(*trace_series*)

Prepare trace_series geometries for a faster spatial analysis.

Assumes geometries are LineStrings which are consequently collected into a single MultiLineString which is prepared with shapely.prepared.prep.

```
>>> traces = gpd.GeoSeries(
...     [LineString([(0, 0), (1, 1)]), LineString([(0, 1), (0, -1)])]
... )
:rtype: :py:class:`~shapely.prepared.PreparedGeometry`
```

```
>>> prepare_geometry_traces(traces).context.wkt
'MULTILINESTRING ((0 0, 1 1), (0 1, 0 -1))'
```

fractopo.general.**pygeos_spatial_index**(*geodataset*)

Get PyGEOSSTRTreeIndex from geopandas dataset.

Parameters
geodataset (Union[GeoDataFrame, GeoSeries]) – Geodataset of which spatial index is wanted.

Return type
PyGEOSSTRTreeIndex

Returns
pygeos spatial index.

Raises
TypeError – If the geodataset sindex attribute was not a pygeos spatial index object.

fractopo.general.**r2_scorer**(*y_true*, *y_predicted*)

Score fit with r2 metric.

Changes the scoring to be best at a value of 0.

Return type
float

fractopo.general.**raise_determination_error**(*attribute*, *verb='determining'*, *determine_target='topology (=branches and nodes)'*)

Raise AttributeError if attribute cannot be determined.

```
>>> try:
...     raise_determination_error("parameters")
...     assert False
... except AttributeError as exc:
...     print(f"{str(exc)[0:20]}...")
...
Cannot determine par...
```

fractopo.general.**random_points_within**(*poly*, *num_points*)

Get random points within Polygon.

```
>>> from pprint import pprint
>>> random.seed(10)
>>> poly = box(0, 0, 1, 1)
>>> result = random_points_within(poly, 2)
:rtype: :py:class:`~typing.List`\[:py:class:`~shapely.geometry.point.Point`]
```

```
>>> pprint([point.within(poly) for point in result])
[True, True]
```

fractopo.general.**read_geofile**(*path*)

Read a filepath for a GeoDataFrame representable geo-object.

Parameters
path (Path) – pathlib.Path to a GeoDataFrame representable spatial file.

Return type
GeoDataFrame

Returns
geopandas.GeoDataFrame read from the file.

Raises
TypeError – If the file could not be parsed as a GeoDataFrame by geopandas.

fractopo.general.**remove_duplicate_caseinsensitive_columns**(*columns*)

Remove duplicate columns case-insensitively.

Return type
set

fractopo.general.**remove_z_coordinates**(*geometry*)

Remove z-coordinates from a geometry.

A shapely geometry should be provided. Output will always be the same type of geometry with the z-coordinates removed.

Parameters
geometry (Union[BaseGeometry, BaseMultipartGeometry]) – Shapely geometry

Return type
Any

Returns
Shapely geometry with the same geometry type

fractopo.general.**remove_z_coordinates_from_geodata**(*geodata*)

Remove Z-coordinates from geometries in geopandasgeodata.

Return type
Union[GeoDataFrame, GeoSeries]

fractopo.general.**replace_coord_in_trace**(*trace*, *index*, *replacement*)

Replace coordinate Point of LineString at index with replacement Point.

Return type
LineString

fractopo.general.**resolve_split_to_ls**(*geom*, *splitter*)

Resolve split between two LineStrings to only LineString results.

Return type
List[LineString]

fractopo.general.**safe_buffer**(*geom*, *radius*, ***kwargs*)

Get type checked Polygon buffer.

```
>>> result = safe_buffer(Point(0, 0), 1)
:rtype: :py:class:`~shapely.geometry.polygon.Polygon`
```

```
>>> isinstance(result, Polygon), round(result.area, 3)
(True, 3.137)
```

fractopo.general.**sanitize_name**(*name*)

Return only alphanumeric parts of name string.

Return type
str

fractopo.general.**save_fig**(*fig*, *results_dir*, *name*)

Save figure as svg image to results dir.

Return type
List[Path]

fractopo.general.**sd_calc**(*data*)

Calculate standard deviation for radial data.

TODO: Wrong results atm. Needs to take into account real length, not just orientation of unit vector. Calculates standard deviation for radial data (degrees)

E.g.

```
>>> sd_calc(np.array([2, 5, 8]))
(3.0, 5.00)
```

Parameters
data (*np.ndarray*) – Array of degrees

Returns
Standard deviation

fractopo.general.**silent_output**(*name*)

General method to silence output from general func.

fractopo.general.**spatial_index_intersection**(*spatial_index*, *coordinates*)

Type-checked spatial index intersection.

Return type
List[int]

fractopo.general.**sum_aggregation**(*values*, **_)

Aggregate by calculating sum.

Return type
Union[float, int]

fractopo.general.**total_bounds**(*geodata*)

Get total bounds of geodataset.

```
>>> geodata = gpd.GeoSeries([Point(-10, 10), Point(10, 10)])
:rtype: :py:data:`~typing.Tuple`\[:py:class:`float`, :py:class:`float`, :py:class:
↪`float`, :py:class:`float`]
```

```
>>> total_bounds(geodata)
(-10.0, 10.0, 10.0, 10.0)
```

fractopo.general.**within_bounds**(*x*, *y*, *min_x*, *min_y*, *max_x*, *max_y*)

Are x and y within the bounds.

```
>>> within_bounds(1, 1, 0, 0, 2, 2)
True
```

fractopo.general.**wrap_silence**(*func*)

Wrap function to capture and silence its output.

Used primarily to silence output from powerlaw functions and methods.

fractopo.general.**write_geodata**(*gdf*, *path*, *driver='GeoJSON'*, *allow_list_column_transform=False*)

Write geodata with driver.

Default is GeoJSON.

fractopo.general.**write_geodataframe**(*geodataframe*, *name*, *results_dir*)

Save geodataframe as GeoPackage, GeoJSON and shapefile.

fractopo.general.**write_topodata**(*gdf*, *output_path*)

Write branch or nodes GeoDataFrame to *output_path*.

fractopo.general.**zip_equal**(**iterables*)

Zip iterables of only equal lengths.

Module contents

fractopo.

Fracture Network Analysis

PYTHON MODULE INDEX

f

INDEX

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z