Read the Docs Template Documentation

Release 1.0

Read the Docs

Apr 17, 2024

CONTENTS

1 What is PyFlowline? 1
- 1.1 Overview . . . 1
- 1.2 Development . . . 1
- 1.3 Objective . . . 1
- 1.4 Target audience . . . 2
- 1.5 Important notice . . . 2

2 Quickstart 3

3 Installation 5
- 3.1 Overview . . . 5
- 3.2 Requirements . . . 5
  - 3.2.1 Option A . . . 5
  - 3.2.2 Option B . . . 6
  - 3.2.3 Visualization . . . 6

4 Data model 7
- 4.1 Basic . . . 7
- 4.2 Spatial references and computational geometry . . . 8
- 4.3 File I/O . . . 8
  - 4.3.1 Configuration files . . . 8
  - 4.3.2 Inputs . . . 11
  - 4.3.3 Outputs . . . 12

5 Algorithm 15
- 5.1 Overview . . . 15
- 5.2 Flowline simplification . . . 15
  - 5.2.1 Dam associate flowline burning . . . 15
  - 5.2.2 Flowline vertex extraction . . . 15
  - 5.2.3 Split flowline . . . 16
  - 5.2.4 Flow direction correction . . . 16
  - 5.2.5 Remove small river . . . 16
  - 5.2.6 Remove braided flowlines . . . 16
  - 5.2.7 Flowline confluence extraction . . . 17
  - 5.2.8 Merge flowline . . . 17
  - 5.2.9 Flowline confluence definition . . . 18
  - 5.2.10 Stream segment index . . . 19
  - 5.2.11 Stream segment order . . . 19
  - 5.2.12 Split flowline by length . . . 19
- 5.3 Mesh generation . . . 19

5.3.1 Structured mesh . . . 19
5.3.2 Unstructured mesh . . . 20
5.4 Topological relationship reconstruction . . . 20
5.4.1 Mesh and flowline intersection . . . 20
5.4.2 Topological relationship reconstruction . . . 20
5.4.3 Remove returning flowline . . . 21

6 Application **23**
6.1 Overview . . . 23
6.2 Model simulation . . . 23
6.2.1 Step 1 . . . 23
6.2.2 Step 2 . . . 23
6.2.3 Step 3 . . . 24
6.2.4 Step 4 . . . 24
6.2.5 Step 5 . . . 24
6.2.6 Step 6 . . . 24
6.2.7 Step 7 . . . 25

7 Visualization **27**

8 References **29**

9 History **31**

10 Authors **33**

11 Support **35**

12 Contribution **37**

13 Frequently Asked Questions **39**

14 API Reference **41**
14.1 Class . . . 41
14.1.1 Basic elements . . . 41
14.1.2 Mesh cell . . . 47
14.1.3 Others . . . 52

15 Addendum **59**
15.1 Glossary . . . 59
15.1.1 Structured mesh . . . 59
15.1.2 Unstructured mesh . . . 59
15.1.3 Great circle . . . 59
15.1.4 DGGS . . . 60
15.1.5 TIN . . . 60
15.1.6 MPAS . . . 60

16 Indices and tables **61**

Index **63**

CHAPTER

ONE

WHAT IS PYFLOWLINE?

1.1 Overview

PyFlowline is a mesh-independent river network generator for hydrologic models.

River networks are landscape features typically represented using vector layers. However, most hydrologic models rely on regular grids to discretize the spatial domain and cannot directly ingest vector features into the model. As a result, hydrologic models usually implement a so-called stream-burning process to convert the vector-based river network into a mesh-based river network.

However, all the existing stream-burning methods only support the structured meshes and there are also some other limitations. For example, existing stream-burning methods always treat the vector river networks as a binary mask and cannot describe the topology near river confluences and meanders.

PyFlowline solves this issue by using a mesh-independent approach that intersects the vector river network and mesh to reconstruct the conceptual river network.

1.2 Development

PyFlowline is developed in an open-source, public repository hosted on Github: https://github.com/changliao1025/pyflowline

1.3 Objective

All the existing river network representation methods (except vector-based) only support the structured rectangle meshes. As a result, if a spatially-distributed hydrologic model uses the unstructured mesh as the spatial discretization, there is no way to represent the river network.

To close this gap, PyFlowline was developed using a mesh-independent approach. At its core, PyFlowline uses the intersection between the vector river network and mesh to reconstruct the conceptual river network.

1.4 Target audience

PyFlowline is an advanced modeling tool for hydrologists and hydrologic modelers. Users of PyFlowline should be familiar with basic concepts in Geographic Information System (GIS), including vector and raster data, coordinate systems, and projections.

1.5 Important notice

1. PyFlowline is designed to run at regional to global scale, so all the datasets use the geographic coordinate system (GCS) with the WGS84 datum. See more details at https://pyflowline.readthedocs.io/en/latest/data/data.html
2. Visualization of the PyFlowline outputs is only experimental. This feature is not fully developed yet. There is ongoing effort to use the *PyEarth* python package to provide this feature.

CHAPTER

TWO

QUICKSTART

Installing and running PyFlowline requires some basic knowledge of the Python ecosystem.

Besides, configuring a PyFlowline simulation requires some knowledge of Geographic Information System (GIS) and computational hydrology.

Users can run a PyFlowline simulation in the following steps:

1. Create a new Python environment using Conda, and activate the new environment.
2. Install the package using *conda install -c conda-forge pyflowline*. Conda will automatically install all the required dependencies.
3. Clone the latest PyFlowline repository from https://github.com/changliao1025/pyflowline.
4. Download the additional large MPAS mesh file *lnd_cull_mesh.nc* from https://github.com/changliao1025/pyflowline/releases/tag/0.2.0 and move it under the *data/susquehanna/input* folder.
5. Open the *examples/susquehanna/pyflowline_susquehanna_mpas.json* file and make the following changes: - Change *sWorkspace_output* to the full path to the directory where you want to save the output (e.g. */full/path/to/pyflowline/data/susquehanna/output*). - Change *"sFilename_mesh_netcdf"* to the full path to *lnd_cull_mesh.nc*. - Change *"sFilename_mesh_boundary"* to the full path to *data/susquehanna/input/boundary_wgs.geojson*. - Change *"sFilename_basins"* to the full path to *examples/susquehanna/pyflowline_susquehanna_basins.json*.
6. Open the *examples/susquehanna/pyflowline_susquehanna_basins.json* file and change *"sFilename_flowline_filter"* to the full path to *data/susquehanna/input/flowline.geojson*. Ignore the other settings in these json files for now.
7. Open the preferred Python IDE (Visual Studio Code recommended) and run the *examples/susquehanna/run_simulation_mpas.py* Python script. Optionally, you can also run the *notebooks/mpas_example.ipynb* notebook. The visualization of the model outputs is only experimental, and you can use other tools to visualize the model outputs.
8. You should produce a list of model outputs in the *data/susquehanna/output* folder or the user-specified output folder.

If you encounter any issues, refer to the FAQ or submit a GitHub issue (https://github.com/changliao1025/pyflowline/issues).

CHAPTER

THREE

INSTALLATION

3.1 Overview

This document provides the instruction to install the PyFlowline Python package.

Two different options are provided below.

3.2 Requirements

We recommend to use the Conda system to install the PyFlowline package.

Conda can be installed on Linux, MacOS, and Windows systems. Please refer to the conda website for details on how to install Conda: https://docs.conda.io/en/latest/

After Conda is available on your system, you can create a conda environment for your application of PyFlowline. Then use Option A or B to install PyFlowline in the newly created environment.

3.2.1 Option A

In this option, you will use conda to install the released PyFlowline package, but not necessarily the latest version. Conda will automatically install all the dependency packages.

Before you install the package, it is highly recommended that you start from a new conda environment using the following command:

```
conda create -n pyflowline_test
```

After activating the environment with:

```
conda activate pyflowline_test
```

You can then install it with:

```
conda install -c conda-forge pyflowline
```

3.2.2 Option B

In this option, you have the opportunity to manually install the *nightly* version.

First, you need to clone the PyFlowline package from GitHub directly.

Navigate into the downloaded folder and manually install the package using:

```
python setup.py install
```

The following dependency packages will be installed during the process.

- *numpy*
- *gdal*
- *netCDF4*

3.2.3 Visualization

PyFlowline only provides experimental support for visualization through the optional *matplotlib* and *cartopy* packages.

You need to manually speicify these packages during the installation process

```
conda install -c conda-forge pyflowline matplotlib cartopy
```

or install manually after the installation of PyFlowline:

```
conda install -c conda-forge matplotlib cartopy
```

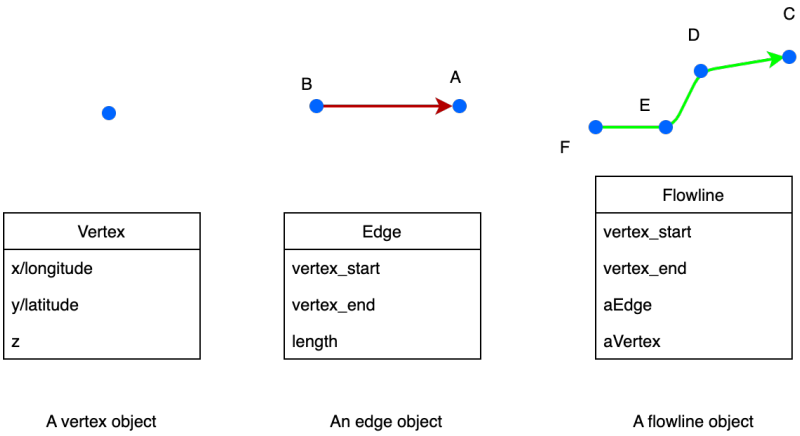
CHAPTER

FOUR

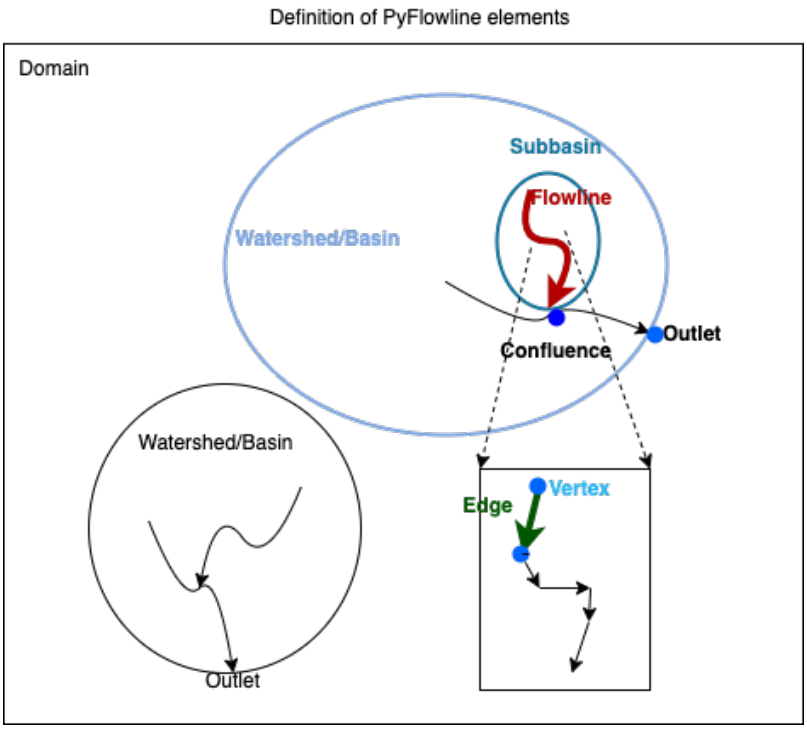
DATA MODEL

4.1 Basic

River networks are represented using three basic elements: vertex, edge, and flowline.



Within PyFlowline, these three elements are combined with several other data structures.



This figure illustrates a domain containing two watersheds/basins. Each basin has an outlet. Within each basin, there are several subbasins and confluences. The lower right is a zoom-in view of a flowline.

4.2 Spatial references and computational geometry

All the internal data elements use the geographic coordinate system (GCS).

All the computational geometry algorithms are based on GCS:

	Input	Output	Algorithm
Location	vertex(lon, lat)	vertex(lon, lat)	
Distance	vertex A, B	Distance (m)	Great circle
Area	vertex A, B, C, … D	Distance (m2)	Spheric area

4.3 File I/O

4.3.1 Configuration files

PyFlowline uses two JSON-format configuration files to manage all input information, where major model input parameters and paths are specified. These configuration files have a parent-child relationship:

1. The parent configuration file stores parameters for the entire domain.
2. The child configuration file stores parameters for each individual watershed.

These files serve as the entry point for setting up and running a PyFlowline case. They can exist wherever the user prefers, but PyFlowline uses the paths specified in these files to locate model inputs and write outputs. Model inputs, outputs, and a recommended directory structure are described in the following two sections.

To create a new PyFlowline case, pass the full path to the parent configuration file to the *pyflowline_read_model_configuration_file* function. This will return a PyFlowline object configured with the values from the file, and it can be used to run the model. See the example notebooks for a demonstration.

Note that the "parent" configuration file contains one block of parameter-value pairs that apply to the entire domain. In contrast, the "child" configuration file contains one block of parameter-value pairs for each watershed. A domain with a single watershed will have a single block in the "child" configuration file, while a domain with multiple watersheds will have multiple blocks.

An example parent JSON file for the Susquehanna River Basin domain (with <domain_name> as "susquehanna") is provided below:

```
{
    "sFilename_model_configuration": "/full/path/to/pyflowline/pyflowline/config/
↪hexwatershed_susquehanna_mpas.json",
    "sWorkspace_data": "/full/path/to/input/data",
    "sWorkspace_output": "/full/path/to/output",
    "sWorkspace_project": "/hexwatershed/susquehanna",
    "sWorkspace_bin": "/full/path/to/bin",
    "sRegion": "susquehanna",
    "sModel": "pyflowline",
    "sJob": "hex",
    "iFlag_standalone": 1,
    "iFlag_create_mesh": 1,
    "iFlag_mesh_boundary": 1,
    "iFlag_save_mesh" :1 ,
    "iFlag_simplification": 1,
    "iFlag_intersect": 1,
    "iFlag_flowline":1,
    "iFlag_use_mesh_dem":1,
    "iFlag_global": 0,
    "iFlag_multiple_outlet": 0,
    "iFlag_rotation": 0,
    "iFlag_mesh_boundary": 0,
    "iCase_index": 1,
    "iMesh_type": 4,
    "dLongitude_left": -79,
    "dLongitude_right": -74.5,
    "dLatitude_bot": 39.20,
    "dLatitude_top": 42.8,
    "dResolution_degree": 5000,
    "dResolution_meter": 5000,
    "sDate": "20220110",
    "sMesh_type": "mpas",
    "sFilename_spatial_reference": "/full/path/to/pyhexwatershed_icom/data/susquehanna/
↪input/boundary_proj_buff.shp",
    "sFilename_dem": "/full/path/to/pyhexwatershed_icom/data/susquehanna/input/dem_buff_
↪ext.tif",
    "sFilename_mesh_netcdf": "/full/path/to/lnd_cull_mesh.nc",
    "sFilename_mesh_boundary": "/full/path/to/pyflowline/data/susquehanna/input/boundary_
↪wgs.geojson",
    "sFilename_basins": "/full/path/to/pyflowline/examples/susquehanna/pyflowline_
↪susquehanna_basins.json"
}
```

Parameter	Data type	Usage	Default value	Note
sFilename_model_configuration	string	The filename of the configuration file	None	Automatically generated
sWorkspace_data	string	The workspace for data	None	Unused
sWorkspace_output	string	The output workspace	None	The output folder
sWorkspace_project	string	The project workspace	None	Unused
sWorkspace_bin	string	The workspace for binary executable	None	Reserved for HexWatershe
sRegion	string	Study region (domain name)	None	None
sModel	string	Model name	pyflowline	None
sJob	string	HPC batch job name	pyflowline	None
iFlag_standalone	int	Flag to run pyflowlone standalone	1	0 when called by hexwater
iFlag_mesh_boundary	int	Flag to use mesh boundary file	1	If 0, use dLongitude/Latitu
iFlag_create_mesh	int	Flag to create mesh	1	None
iFlag_save_mesh	int	Flag to save mesh	1	None
iFlag_simplification	int	Flag to simplification	1	None
iFlag_intersect	int	Flag to intersect	1	None
iFlag_flowline	int	Flag for flowline	1	None
iFlag_use_mesh_dem	int	Flag to use DEM data	0	Not used
iFlag_global	int	Flag to run on global scale	0	None
iFlag_multiple_outlet	int	Flag to run with multi-outlet	0	None
iFlag_rotation	int	Flag for hexagon rotation	0	None
iCase_index	int	Index of case	1	None
iMesh_type	int	Type of mesh	1	None
dLongitude_left	float	Boundary	-180	None
dLongitude_right	float	Boundary	+180	None
dLatitude_bot	float	Boundary	-90	None
dLatitude_top	float	Boundary	+90	None
dResolution_degree	float	Resolution in degree	1	None
dResolution_meter	float	Resolution in meter	5000	None
sDate	string	Date of simulation	None	None
sMesh_type	string	Mesh type	None	None
sFilename_spatial_reference	string	Spatial reference	None	None
sFilename_dem	string	DEM file	None	Reserved for HexWatershe
sFilename_mesh_netcdf	string	Netcdf mesh file	None	
sFilename_mesh_boundary	string	Domain boundary file	None	Required if iFlag_mesh_bo
sFilename_basins	string	Filename of child JSON file	None	None

An example child JSON file is provided below:

```
[
{
    "dLatitude_outlet_degree": 39.4620,
    "dLongitude_outlet_degree": -76.0093,
    "dAccumulation_threshold": 100000,
    "dThreshold_small_river": 10000,
    "iFlag_dam": 0,
    "iFlag_debug":1,
    "iFlag_disconnected": 0,
    "lBasinID": 1,
    "sFilename_dam": "/full/path/to/hexwatershed/susquehanna/auxiliary/dams.csv",
    "sFilename_flowline_filter": "/full/path/to/pyflowline/data/susquehanna/input/
↪flowline.geojson",
```

(continues on next page)

(continued from previous page)

```
    "sFilename_flowline_raw": "/full/path/to/hexwatershed/susquehanna/vector/hydrology/
↪allflowline.shp",
    "sFilename_flowline_topo": "/full/path/to/hexwatershed/susquehanna/auxiliary/
↪flowline.csv"
}
]
```

Parameter	Data type	Usage	Default value	Note
dLatitude_outlet_degree	float	The latitude of outlet	None	
dLongitude_outlet_degree	float	The longitude of outlet		
dAccumulation_threshold	float	The flow accumulation threshold		
dThreshold_small_river	float	The small river threshold		
iFlag_dam	int	Flag for dam burning	0	
iFlag_debug	int	Flag to turn on debug info	0	
iFlag_disconnected	int	Flag for disconnected flowline	0	
lBasinID	int	Basin/watershed ID	0	
sFilename_dam	string	Filename of dam file	1	Only used for dam burning
sFilename_flowline_filter	string	Filename of original flowline file		GeoJSON format
sFilename_flowline_raw	string	Filename of flowline including dam		Only used for dam burning
sFilename_flowline_topo	string	Filename of dam topology		Only used for dam burning

4.3.2 Inputs

The following recommended workspace structure and example input files are provided to run a PyFlowline simulation. Although the repo includes example configuration files in the examples/ directory, they can be placed wherever the user prefers, as long as the paths within them point to the correct locations for input (and output) data.

```
data
└── <domain_name>
    ├── input
    │   ├── boundary_wgs.geojson
    │   ├── flowline.geojson
    │   ├── pyflowline_<domain_name>_<meshtype>.json
    │   └── pyflowline_<domain_name>_basins.json
    └── output
        └── ...
```

4.3.3 Outputs

After running the PyFlowline simulation, the output workspace will be structured as follows:

```
data
└── <domain_name>
        ├── input
        │   ...
        └── output
            └── <pyflowline_casename>
                ├── 00000001
                │   ├── basin_info.json
                │   ├── confluence_conceptual_info.json
                │   ├── confluence_simplified_info.json
                │   ├── flowline_conceptual.json
                │   ├── flowline_conceptual_info.json
                │   ├── flowline_edge.json
                │   ├── flowline_filter.json
                │   ├── flowline_intersect_mesh.json
                │   ├── flowline_simplified.json
                │   ├── flowline_simplified_info.json
                │   └── vertex_simplified.json
                │       ├── 00000002
                │   ├── basin_info.json
                │   ├── confluence_conceptual_info.json
                │       │   └── ...
                ├── mpas.json
                ├── mpas_mesh_info.json
                    ├── run_pyflowline.py
                    ├── submit.job
                    ├── stdout.out
                    └── stderr.err
```

The sub-folders *00000001* et. al, are results for every watershed. Within each watershed sub-folder, there are both json and geojson model output files. The primary (and final) PyFlowline model-generated flowline is *flowline_conceptual.json*. This file is in the GEOJSON format, and can be viewed directly in QGIS or similar software. Other files that may be of particular interest to users include the model-generated mesh file *mpas_mesh_info.json* which contains a complete description of the model-generated mesh, and *mpas.json* which contains the same information in the GEOJSON format, and can be viewed directly in QGIS or similar software. In the <pyflowline_casename> root directory, three HPC-associated files *submit.job*, *stdout.out*, *stderr.err* are generated. The script *run_pyflowline.py* is the python script that was ran by the HPC job. If you are running on a local machine, you can run this script directly. The table below describes the output files.

Filename	Description
basin_info.json	Basin configuration information output file.
confluence_conceptual_info.json	Complete description of conceptual flowline confluence nodes.
confluence_simplified_info.json	Complete description of simplified flowline confluence nodes.
flowline_conceptual.json	Final modeled flowline in GEOJSON format.
flowline_conceptual_info.json	Final modeled flowline in JSON format.
flowline_edge.json	
flowline_filter.json	
flowline_intersect_mesh.json	Intermediate modeled flowline in GEOJSON format.
flowline_simplified.json	Intermediate modeled flowline in GEOJSON format.
flowline_simplified_info.json	Intermediate modeled flowline in JSON format.
vertex_simplified.json	Flowline vertex file in GEOJSON format.
mpas.json	Model generated mesh file in GEOJSON format. Contains complete mesh description.
mpas_mesh_info.json	Model generated mesh file in JSON format. Contains complete mesh description.
run_pyflowline.py	Python script that was run by the HPC job (can be run directly on a local machine).
submit.job	HPC associated file
stdout.out	HPC associated file
stderr.err	HPC associated file

CHAPTER

FIVE

ALGORITHM

5.1 Overview

A list of algorithms is implemented to carry out the following operations:

1. Flowline simplification
2. Mesh generation
3. Topological relationship reconstruction

5.2 Flowline simplification

5.2.1 Dam associate flowline burning

(Optional)

Through a look-up table that links dams with their associated flowlines, this algorithm includes all the downstream flowlines of each dam into the flowline simplification process.

Currently, this algorithm does not include the upstream of a dam.

5.2.2 Flowline vertex extraction

The vertices that make up flowlines are used in several algorithms. Among them, a flowline's starting and ending vertices also define the flowline type.

- If the starting vertex has no upstream, the flowline is a headwater.
- If the starting or ending vertex has only one upstream or downstream, it is a middle flowline and can be merged with others.
- If a starting vertex has more than one upstream vertices, it is a river confluence.

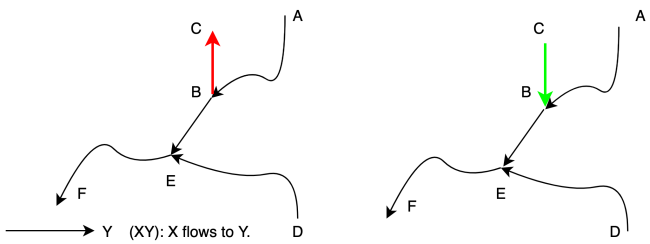
5.2.3 Split flowline

With all the flowlines and vertices, the algorithm split the flowlines into a minimal set that meets the following requirement:

- All flowlines' starting and ending vertices are made up by the vertex loop-up table.
- No flowline has a middle vertex that belongs to the same look-up table.

5.2.4 Flow direction correction

Due to data quality issues, the existing flowlines may have incorrect flow directions, which lead to multiple downstream flow directions. The corresponding node connection matrix has rows with multiple *1*s. This algorithm scans from the outlet vertex and searches reversely; once such a row is detected, the corresponding flow direction is reversed.



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	0	0	0
D	0	0	0	0	1	0
E	0	0	0	0	0	1
F	0	0	0	0	0	0

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	1	0
C	0	1	0	0	0	0
D	0	0	0	0	1	0
E	0	0	0	0	0	1
F	0	0	0	0	0	0

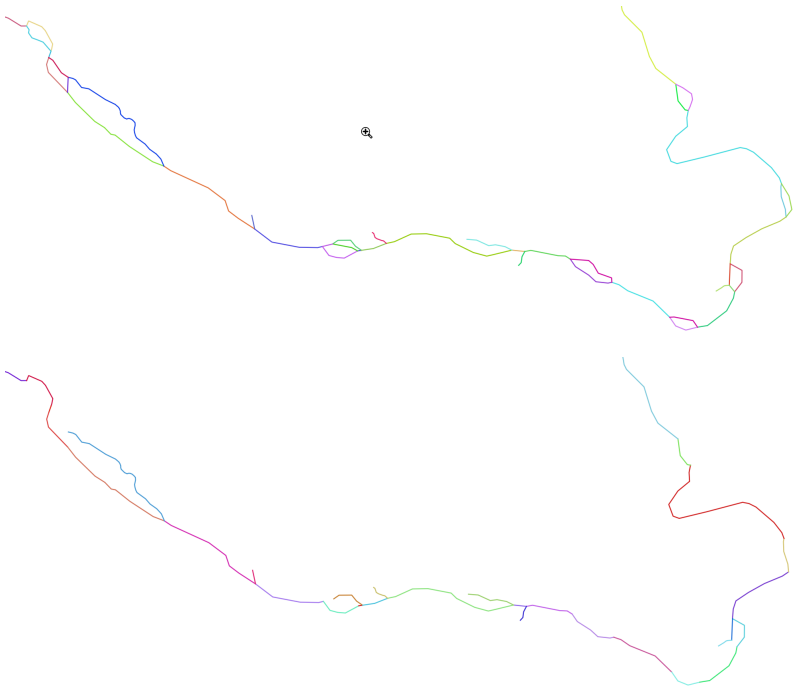
5.2.5 Remove small river

To simplify the river networks, small rivers with lengths less than the user-provided threshold are removed. This algorithm only applies to headwater and should be called multiple times to achieve desired performance.

(Optional) When the dam burning is turned on, the dam-associated flowlines are always retained even if their lengths are less than the user-provided threshold.

5.2.6 Remove braided flowlines

A braided loop occurs when a vertex has more than one downstream, even after the flow direction correction. This algorithm removes these loops by only keeping the first detected downstream of any vertex.



5.2.7 Flowline confluence extraction

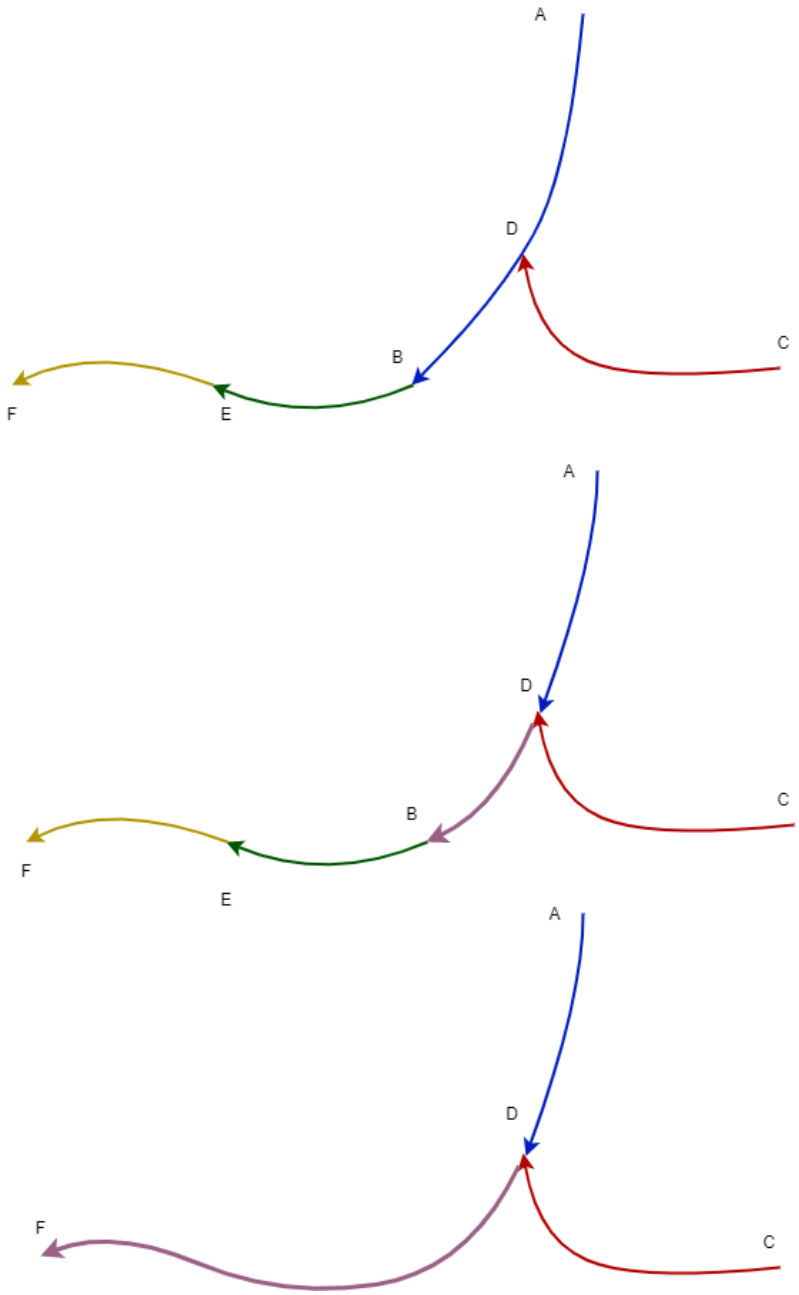
This algorithm scans the whole network and defines the vertices that have more than one upstream flowline as river confluences.

5.2.8 Merge flowline

This algorithm merges flowlines, so there are only two types of flowlines:

1. headwaters
2. flowline between the confluences

If there are multiple flowlines within the same confluence bound, they are merged as one.



5.2.9 Flowline confluence definition

After the flowlines are in the final format, the confluences are redefined using the same criteria as above.

5.2.10 Stream segment index

This algorithm defines the stream segment index using their topologic relationship..

5.2.11 Stream segment order

This algorithm defines the stream order based on the stream segment topology and the classic stream order, also called Hack's stream order or Gravelius' stream order method.

5.2.12 Split flowline by length

(Optional)

In some cases, it is desirable to impose a maximum flowline edge length so it can be used in other applications. This algorithm divides such kinds of edges until they meet the requirement.

5.3 Mesh generation

PyFlowline provides several algorithms to generate structured meshes, including latitude-longitude, projected, hexagon, triangle meshes.

The hexagon mesh generator also provides an option for a 60-degree rotation (https://www.redblobgames.com/grids/hexagons/#basics).

PyFlowline uses the geographic coordinate system (GCS) exclusively for all the computational geometry, all the meshes are converted to the GCS system. See the hexagon mesh for an example.

5.3.1 Structured mesh

In general, the mesh generator creates mesh cells one by one in a pre-defined row-column order starting from the lower left corner. The generator calculates the locations of all vertices of each mesh cell. The coordinates will be converted back to GCS if the mesh is in PCS.

Usually, the domain's boundary is defined in the configuration file, and the algorithm starts from the lower left.

Latitude-longitude mesh

1. Coordinates of 4 vertices are calculated, then a cell is defined.
2. Repeat until all cells are generated

Projected mesh

1. Coordinates of 4 vertices are calculated and re-projected to GCS, then a cell is defined.
2. Repeat until all cells are generated

Hexagon mesh

1. Coordinates of 6 vertices are calculated and re-projected to GCS, then a cell is defined.
2. Repeat until all cells are generated

Triangle mesh

1. Coordinates of 3 vertices are calculated and re-projected to GCS, then a cell is defined.
2. Repeat until all cells are generated

5.3.2 Unstructured mesh

PyFlowline does not provide unstructured mesh generations. Instead, the user should use third-party generators such as the JIGSAW to generate the mesh files. PyFlowline only provides algorithms to import these mesh files and convert them to PyFlowline-supported mesh data type.

MPAS

Supported by *JIGSAW*

TIN

Not yet supported

DGGrid

Not yet supported

5.4 Topological relationship reconstruction

5.4.1 Mesh and flowline intersection

This algorithm calls the GDAL (https://gdal.org/) APIs to intersect the mesh with the simplified river network. Each stream segment is broken into reaches.

5.4.2 Topological relationship reconstruction

After the intersection, this algorithm rebuilds the topologic relationship using the entrance and exit vertices of each reach to construct the reach-based or cell center-based river network.

5.4.3 Remove returning flowline

This algorithm simplifies the topology information for several unusual scenarios. For example, if a flowline leaves and reenters the same mesh cell through the same edge, this creates a loop in topology and will be simplified.

CHAPTER

SIX

APPLICATION

6.1 Overview

An example is provided within the *examples* folder. This example contains a case study in the *Susquehanna* watershed with several python scripts (corresponding to four mesh types). For example, the MPAS mesh type-based case is explained here.

6.2 Model simulation

6.2.1 Step 1

The example *run_simulation_mpas.py* script import a few packages and functions.

```
import os, sys
from pathlib import Path
from os.path import realpath
from pyflowline.pyflowline_read_model_configuration_file import pyflowline_read_model_
↪configuration_file
```

The *pyflowline_read_model_configuration_file* function reads in a JSON configuration file and loads all the necessary model parameters.

6.2.2 Step 2

The script sets up some paths, which should be adjusted based on a real case and your local directory structure.

```
sPath_parent = str(Path(__file__).parents[2]) # data is located two dir's up
sPath_data = realpath( sPath_parent +  '/data/susquehanna' )
sWorkspace_input =  str(Path(sPath_data)  /  'input')
sWorkspace_output=  str(Path(sPath_data)  /  'output')
sWorkspace_output=  '/full/path/to/pyflowline/data/susquehanna/output'
```

6.2.3 Step 3

Check the configuration file:

```
sFilename_configuration_in = realpath( sPath_parent +  '/examples/susquehanna/pyflowline_
↪susquehanna_mpas.json' )
if os.path.isfile(sFilename_configuration_in):
    pass
else:
    print('This configuration does not exist: ', sFilename_configuration_in )
```

6.2.4 Step 4

Set up case information and read the configuration file.

```
iCase_index = 17
Mesh = 'mpas'
Date='20220901'


oPyflowline = pyflowline_read_model_configuration_file(sFilename_configuration_in, \
   iCase_index_in=iCase_index, sDate_in=sDate)
oPyflowline.aBasin[0].dLatitude_outlet_degree=39.462000
oPyflowline.aBasin[0].dLongitude_outlet_degree=-76.009300
```

6.2.5 Step 5

Setup the model and run the three major steps.

```
oPyflowline.setup()
oPyflowline.flowline_simplification()
Cell = oPyflowline.mesh_generation()
oPyflowline.reconstruct_topological_relationship(aCell)
```

6.2.6 Step 6

Analyze and export the model outputs.

```
oPyflowline.analyze()
oPyflowline.evaluate()
oPyflowline.export()
```

6.2.7 Step 7

Optionally, the user can also visualize the model outputs using the following method.

```
aExtent_full = [-78.5,-75.5, 39.2,42.5]
sFilename =  'filtered_flowline.png'
oPyflowline._plot(sFilename, sVariable_in = 'flowline_filter', aExtent_in =aExtent_full )

sFilename =  'conceptual_flowline_with_mesh.png'
oPyflowline._plot(sFilename,  iFlag_title=1 ,sVariable_in='overlap',   aExtent_in=aExtent_full )
```

CHAPTER

SEVEN

VISUALIZATION

The built-in visualization feature is experimental. Currently, users are recommended to use third-party tools such as QGIS to visualize the GEOJSON outputs.

CHAPTER

EIGHT

REFERENCES

- Liao. C. Cooper, M (2022) Pyflowline: a mesh-independent river network generator for hydrologic models. Zenodo.

https://doi.org/10.5281/zenodo.6407299

CHAPTER

NINE

HISTORY

- 2020-10-01: Design

CHAPTER

TEN

AUTHORS

- Chang Liao (Pacific Northwest National Laboratory)
- Matt G Cooper (Pacific Northwest National Laboratory)

CHAPTER

ELEVEN

SUPPORT

Support is provided through Github issue(https://github.com/changliao1025/pyflowline/issues).

CHAPTER

TWELVE

CONTRIBUTION

PyFlowline was developed and maintained by

- Chang Liao (Pacific Northwest National Laboratory)

CHAPTER

THIRTEEN

FREQUENTLY ASKED QUESTIONS

1. Why my *conda* cannot create environment?

   Turn off the VPN or bypass it.

2. Why import *GDAL* failed?

   Consider using the *conda-forge* channel.

3. *proj* related issue https://github.com/OSGeo/gdal/issues/1546,

   Make sure you correctly set up the *PROJ_LIB*

   Because the *GDAL* library is used by this project and the *proj* library is often not configured correctly automatically. On Linux or Mac, you can set it up using the *.bash_profile* such as:

   Anaconda:

   export PROJ_LIB=/people/user/.conda/envs/hexwatershed/share/proj

   export PROJ_LIB=$HOME/opt/anaconda3/envs/pyflowline/share/proj

   Miniconda:

   export PROJ_LIB=/opt/miniconda3/envs/hexwatershed/share/proj

4. I am getting errors using the plot functions

   If you receive an error related to *GeoAxesSubplot*, make sure you have cartopy version 0.21.0 installed in your environment. Optionally, downgrade matplotlib to 3.5.2.

5. What if my model doesn't produce the correct or expected answer?

   Answer: There are several hidden assumptions within the workflow. For example, if you provide the DEM and river network for two different regions, the program won't be able to tell you that. A visual inspection of your data is important.

   Optionally, you can turn on the *iFlag_debug* option in the configuration file to output the *intermediate files*.

CHAPTER

FOURTEEN

API REFERENCE

14.1 Class

14.1.1 Basic elements

pyflowline.classes.vertex.pyvertex : public object

```
The vertex class

Args:
    object (_type_): None

Returns:
    pyvertex: A vertex object
```

Public Functions

__init__(*self*, *aParameter*)

```
Initilize a vertex object

Args:
    aParameter (dict): A dictionary parameters
```

toNvector(*self*)

```
Note: replicated in LatLon_NvectorEllipsoidal

Returns:
    pynvector: A nvector object
```

__eq__(*self*, *other*)

```
Check whether two vertices are equivalent

Args:
    other (pyvertex): The other vertex
```

(continues on next page)

(continued from previous page)

```
Returns:
    int: 1 if equivalent, 0 if not
```

__ne__(*self*, *other*)

```
Check whether two vertices are equivalent

Args:
    other (pyvertex): The other vertex

Returns:
    int: 0 if equivalent, 1 if not
```

calculate_distance(*self*, *other*)

```
Calculate the distance between two vertices

Args:
    other (pyvertex): The other vertex

Returns:
    float: The great circle distance
```

tojson(*self*)

```
Convert a vecter object to a json string

Returns:
    json str: A json string
```

pyflowline.classes.edge.pyedge : public object

```
The pyedge class

Args:
    object (object): None

Returns:
    pyedge: A edge object
```

Public Functions

__init__(*self*, *pVertex_start_in*, *pVertex_end_in*)

```
Initilize a pyedge object

Args:
    pVertex_start_in (pyvertex): The starting vertex
    pVertex_end_in (pyvertex): The ending vertex
```

calculate_length(*self*)

```
Calcualate the length of the edge

Returns:
    float: The length of the edge
```

check_shared_vertex(*self*, *other*)

```
Check whether two edges are sharing the same vertex

Args:
    other (pyedge): The other edge object to be checked

Returns:
    int: Flag, 1: shared; 0: non-sharing
```

check_upstream(*self*, *other*)

```
Check whether another edge is the upstream of current edge

Args:
    other (pyedge): The other edge object to be checked

Returns:
    int: Flag, 1: upstream; 0: non-upstream
```

check_downstream(*self*, *other*)

```
Check whether another edge is the downstream of current edge


Args:
    other (pyedge): The other edge object to be checked

Returns:
    int: Flag, 1: downstream; 0: non-downstream
```

split_by_length(*self*, *dLength_in*)

```
Split an edge using the threshold

Args:
    dLength_in (float): The length threshold

Returns:
    list [pyedge]: A list of edge objects, length of 1 if it meets the␣
↪requirement
```

reverse(*self*)

```
Reverse an edge
```

is_overlap(*self*, *pEdge_in*)

```
Check if two edges overlap each other

Args:
    pEdge_in (pyedge): The other edge to be checked

Returns:
    int: 1 if overlap; 0 if not
```

check_vertex_on_edge(*self*, *pVertex_in*)

```
Check if a vertex on an edge

Args:
    pVertex_in (pyvertex): The vertex to be checked

Returns:
    tuple[int, float, float]: 1 if it is on; 0 if not. Length and distance are␣
↪calculated if on.
```

__eq__(*self*, *other*)

```
Check if two edges are equivalent

Args:
    other (pyedge): The other edge
    how about direction?

Returns:
    int: 1 if equivalent; 0 if not
```

__ne__(*self*, *other*)

```
Check if two edges are equivalent

Args:
    other (pyedge): The other edge

Returns:
    int: 0 if equivalent; 1 if not
```

tojson(*self*)

```
Convert an edge object to a json string

Returns:
    json str: A json string
```

pyflowline.classes.flowline.pyflowline : public object

```
The pyflowline class

Args:
    object (object): None

Returns:
    pyflowline: The flowline object
```

Public Functions

__init__(*self*, *aEdge*)

```
Initilize a flowline object

Args:
    aEdge (list [pyedge]): A list of edge objects
```

calculate_length(*self*)

```
Calcualte the length

Returns:
    float: The length of the flowline
```

check_upstream(*self*, *other*)

```
Check whether another flowline is upstream or not

Args:
    other (pyflowline): The other flowline

Returns:
    int: 1 if it is, 0 if not
```

check_downstream(*self*, *other*)

```
Check whether another flowline is downstream or not

Args:
    other (pyflowline): The other flowline

Returns:
    int: 1 if it is, 0 if not
```

reverse(*self*)

```
Reverse a flowline
```

merge_upstream(*self*, *other*)

```
Merge two flowlines as one

Args:
    other (pyflowline): The other flowline

Returns:
    pyflowline: The merged flowline
```

split_by_length(*self*, *dDistance*)

```
Split a flowline using the length threshold

Args:
    dDistance (float): The length threshold for each edge

Returns:
    pyflowline: The updated flowline
```

calculate_flowline_sinuosity(*self*)

```
Calculate the sinuosoty of a flowline
```

__eq__(*self*, *other*)

```
Check whether two flowline are equivalent

Args:
    other (pyflowline): The other flowline

Returns:
    int: 1 if equivalent, 0 if not
```

__ne__(*self*, *other*)

```
Check whether two flowline are equivalent

Args:
    other (pyflowline): The other flowline

Returns:
    int: 0 if equivalent, 1 if not
```

tojson(*self*)

```
Convert a pyflowline object to a json string

Returns:
    json str: A json string
```

14.1.2 Mesh cell

pyflowline.classes.hexagon.pyhexagon : public pyflowline.classes.cell.pycell

```
The hexagon cell class

Args:
    pycell (obj): None

Returns:
    pyhexagon: A hexagon object
```

Public Functions

__init__(*self*, *dLon*, *dLat*, *aEdge*, *aVertex*)

```
Initilize a hexagon cell object

Args:
    dLon (float): The longitude of center
    dLat (float): The latitude of center
    aEdge (list [pyedge]): A list of edges that define the hexagon
    aVertex (list [pyvertex]): A list of vertices the define the hexagon
```

has_this_edge(*self*, *pEdge_in*)

```
Check whether the hexagon contains an edge

Args:
    pEdge_in (pyedge): The edge to be checked

Returns:
    int: 1 if found, 0 if not
```

which_edge_cross_this_vertex(*self*, *pVertex_in*)

```
Find which edge overlap with a vertex

Args:
    pVertex_in (pyvertex): The vertex to be checked

Returns:
    tuple [int, pyedge]: 1 if found, with the edge object; 0 if not found
```

calculate_cell_area(*self*)

```
Calculate the area of the hexagon cell

Returns:
    float: The area in m2
```

calculate_edge_length(*self*)

```
Calculate the effective length of the hexagon cell

Returns:
    float: The effective length
```

share_edge(*self*, *other*)

```
Check whether a hexagon shares an edge with another hexagon

Args:
    other (pyhexagon): The other hexagon

Returns:
    int: 1 if share, 0 if not
```

tojson(*self*)

```
Convert a hexagon object to a json string

Returns:
    json str: A json string
```

pyflowline.classes.square.pysquare : public pyflowline.classes.cell.pycell

```
The square cell class

Args:
    pycell (_type_): None

Returns:
    pysquare: A square cell object
```

Public Functions

__init__(*self*, *dLon*, *dLat*, *aEdge*, *aVertex*)

```
Initilize a square cell object

Args:
    dLon (float): The longitude of center
    dLat (float): The latitude of center
    aEdge (list [pyedge]): A list of edges that define the square cell
    aVertex (list [pyvertex]): A list of vertices the define the square cell
```

has_this_edge(*self*, *pEdge_in*)

```
Check whether the square contains an edge

Args:
```

(continues on next page)

(continued from previous page)

```
    pEdge_in (pyedge): The edge to be checked

Returns:
    int: 1 if found, 0 if not
```

which_edge_cross_this_vertex(*self*, *pVertex_in*)

```
Find which edge overlap with a vertex

Args:
    pVertex_in (pyvertex): The vertex to be checked

Returns:
    tuple [int, pyedge]: 1 if found, with the edge object; 0 if not found
```

calculate_cell_area(*self*)

```
Calculate the area of the hexagon cell

Returns:
    float: The area in m2
```

calculate_edge_length(*self*)

```
Calculate the effective length of the square cell

Returns:
    float: The effective length
```

share_edge(*self*, *other*)

```
Check whether a square cell shares an edge with another cell

Args:
    other (pysquare): The other cell

Returns:
    int: 1 if share, 0 if not
```

tojson(*self*)

```
Convert a square object to a json string

Returns:
    json str: A json string
```

pyflowline.classes.latlon.pylatlon : public pyflowline.classes.cell.pycell

```
The latlon cell class

Args:
```

(continues on next page)

(continued from previous page)

```
    pycell (obj): None

Returns:
    pylatlon: A latlon cell object
```

Public Functions

__init__(*self*, *dLon*, *dLat*, *aEdge*, *aVertex*)

```
Initilize a latlon cell object

Args:
    dLon (float): The longitude of center
    dLat (float): The latitude of center
    aEdge (list [pyedge]): A list of edges that define the latlon cell
    aVertex (list [pyvertex]): A list of vertices the define the latlon
```

has_this_edge(*self*, *pEdge_in*)

```
Check whether the latlon contains an edge

Args:
    pEdge_in (pyedge): The edge to be checked

Returns:
    int: 1 if found, 0 if not
```

which_edge_cross_this_vertex(*self*, *pVertex_in*)

```
Find which edge overlap with a vertex

Args:
    pVertex_in (pyvertex): The vertex to be checked

Returns:
    tuple [int, pyedge]: 1 if found, with the edge object; 0 if not found
```

calculate_cell_area(*self*)

```
Calculate the area of the latlon cell

Returns:
    float: The area in m2
```

calculate_edge_length(*self*)

```
Calculate the effective length of the latlon cell

Returns:
    float: The effective length
```

share_edge(*self*, *other*)

```
Check whether a latlon shares an edge with another latlon

Args:
    other (pylatlon): The other latlon cell

Returns:
    int: 1 if share, 0 if not
```

tojson(*self*)

```
Convert a latlon object to a json string

Returns:
    json str: A json string
```

pyflowline.classes.mpas.pympas : public pyflowline.classes.cell.pycell

```
The MPAS cell class

Args:
    pycell (object): None

Returns:
    pympas: A mpas cell object
```

Public Functions

__init__(*self*, *dLon*, *dLat*, *aEdge*, *aVertex*)

```
Initilize a mpas cell object

Args:
    dLon (float): The longitude of center
    dLat (float): The latitude of center
    aEdge (list [pyedge]): A list of edges that define the hexagon
    aVertex (list [pyvertex]): A list of vertices the define the hexagon
```

has_this_edge(*self*, *pEdge_in*)

```
Check whether the cell contains an edge

Args:
    pEdge_in (pyedge): the to be checked edge

Returns:
    int: 1 if contains; or else 0
```

which_edge_cross_this_vertex(*self*, *pVertex_in*)

```
When a flowline intersects with a cell, this function finds out which edge is␣
↪intersected

Args:
    pVertex_in (pyvertex): the intersected vertex

Returns:
    tuple: (1, edge) if contains; or else (0, None)
```

calculate_cell_area(*self*)

```
Calculate the area of a cell, this function is not used for mpas cell

Returns:
    float: cell area
```

calculate_edge_length(*self*)

```
Calculate the effective cell length/resolution

Returns:
    float: effective cell length/resolution
```

share_edge(*self*, *other*)

```
Check if two cells share an edge

Args:
    other (pympas): the other cell

Returns:
    int: 1 if shared, 0 if not
```

tojson(*self*)

```
Convert a cell into a json string

Returns:
    json str: A json string
```

14.1.3 Others

pyflowline.classes.basin.pybasin : public object

```
Basin class

Args:
    (object): None

Returns:
    None: A basin object
```

Public Functions

__init__(*self*, *aParameter*)

```
Initialize the basin class object

Args:
    aParameter (dict): Dictionary for parameters
```

basin_flowline_simplification(*self*)

```
Run the basin flowline simplification

Returns:
    list [pyflowline]: A list of simplified flowline
```

basin_reconstruct_topological_relationship(*self*, *iMesh_type*, *sFilename_mesh*)

```
Run the basin topologic relationship reconstruction

Args:
    iMesh_type (int): Mesh type
    sFilename_mesh (str): Filename of the geojson mesh

Returns:
    list [pyflowline]: A list of intersected cells
```

basin_build_confluence(*self*, *aFlowline_basin_in*, *aVertex_confluence_in*)

```
Build the conflence

Args:
    aFlowline_basin_in (list [pyflowline]): A list of flowlines in this basin
    aVertex_confluence_in (list [pyconfluence]): A list of vertices in this␣
↪basin

Returns:
    list [pyconfluence]: A list of confluences in this basin
```

basin_analyze(*self*)

```
Analyze the basin results including length, sinuosity, and breaching angle
```

basin_export(*self*)

```
Export the basin outputs in json format
```

basin_export_flowline(*self*, *aFlowline_in*, *sFilename_json_in*, *iFlag_projected_in=None*, *pSpatial_reference_in=None*)

```
Export the basin flowline to geojson

Args:
```

(continues on next page)

(continued from previous page)

```
    aFlowline_in (list [pyflowline]): A list of flowlines
    sFilename_json_in (str): The output json filename
    iFlag_projected_in (int, optional): Flag if re-projection is needed.␣
↪Defaults to None.
    pSpatial_reference_in (object, optional): The spatial reference if re-
↪projection is needed. Defaults to None.
```

basin_export_basin_info_to_json(*self*)

```
Export the basin basin object to json
```

basin_export_flowline_info_to_json(*self*)

```
Export the flowline object to json
```

basin_export_confluence_info_to_json(*self*)

```
Export the confluence object to json
```

tojson(*self*)

```
Export the basin object to json

Returns:
    json str: A json string
```

basin_export_config_to_json(*self*, *sFilename_output_in=None*)

```
Export the basin object to json using the encoder

Args:
    sFilename_output_in (str, optional): The json filename. Defaults to None.
```

basin_convert_flowline_to_geojson(*self*)

```
Convert the flowline to geojson
```

basin_calculate_flowline_length(*self*, *aFlowline_in*)

```
Calculate the length of flowlines

Args:
    aFlowline_in (list [pyflowline]): A list of flowlines

Returns:
    float: The total length of all flowlines
```

basin_calculate_river_sinuosity(*self*)

```
Calcualte the the river sinuosity
```

basin_calculate_confluence_branching_angle(*self*)

```
Calcualte the the river confluence branching angle
```

basin_evaluate(*self*, *iMesh_type*, *sMesh_type*)

```
Evaluate the model performance

Args:
    iMesh_type (int): The mesh type
    sMesh_type (str): The mesh type
```

basin_evaluate_area_of_difference(*self*, *iMesh_type*, *sMesh_type*)

```
Evaluate the model performance using area of difference

Args:
    iMesh_type (int): The mesh type
    sMesh_type (str): The mesh type
```

pyflowline.classes.pycase.flowlinecase : public object

```
The flowline case class

Args:
    object (obj): None

Returns:
    flowlinecase: A flowlinecase object
```

Public Functions

__init__(*self*, *aConfig_in*, *iFlag_standalone_in=None*, *sModel_in=None*, *sDate_in=None*, *sWorkspace_output_in=None*)

```
Initialize a flowlinecase object

Args:
    aConfig_in (dict): A dictionary of parameters
    iFlag_standalone_in (int, optional): Flag for whether run the case␣
↪standalone. Defaults to None.
    sModel_in (str, optional): The model name. Defaults to None.
    sDate_in (str, optional): The case date. Defaults to None.
    sWorkspace_output_in (str, optional): The output workspace. Defaults to␣
↪None.
```

pyflowline_mesh_generation(*self*, *iFlag_antarctic_in=None*)

```
The mesh generation operation

Returns:
    list [pycell]: A list of cell object
```

pyflowline_reconstruct_topological_relationship(*self*)

```
The topological relationship reconstruction operation

Args:
    aCell_raw (list [pycell]): A list of intersected cell objects

Returns:
    tuple [list [pycell], list [pyflowline], list [long]]: A list of cells,␣
↪flowlines, and outlet cell IDs.
```

pyflowline_merge_cell_info(*self*, *aCell_raw*)

```
Merge cell information after reconstruction

Args:
    aCell_raw (list [pycell]): The original cell information that contains␣
↪neighbor definition
    This information is defined in the mesh generation function, so mesh␣
↪generation must be run.

Returns:
    list [pycell]: The updated list of cell objects.
```

pyflowline_analyze(*self*)

```
Analyze the domain results for every watershed
```

pyflowline_setup(*self*)

```
Set up the flowlinecase
```

pyflowline_run(*self*)

```
Run the flowlinecase simulation

Returns:
    list: A list of cell objects
```

pyflowline_evaluate(*self*)

```
Evaluate the model performance
```

pyflowline_export(*self*)

```
Export the model outputs
```

pyflowline_export_mesh_info_to_json(*self*)

```
Export the mesh information to a json file
```

tojson(*self*)

```
Convert the flowline case object to a json string

Returns:
    json str: A json string
```

pyflowline_print(*self*)

```
Print the flowline case object
```

pyflowline_export_config_to_json(*self*, *sFilename_output_in=None*)

```
Export the configuration to a json file

Args:
    sFilename_output_in (str, optional): The json filename. Defaults to None.
```

pyflowline_export_basin_config_to_json(*self*, *sFilename_output_in=None*)

```
Export the member basin configuration to a json file

Args:
    sFilename_output_in (str, optional): The json filename. Defaults to None.
```

class pyconfluence

```
The pyconfluence class
Returns:
    object: A confluence object
```

Public Functions

__init__(*self*, *pVertex_center*, *aFlowline_upstream_in*, *pFlowline_downstream_in*)

```
Initialize a pyconfluence object

Args:
    pVertex_center (pyvertex): The center vertex
    aFlowline_upstream_in (list [pyflowline]): A list of upstream flowlines
    pFlowline_downstream_in (pyflowline): The downstream flowline
```

calculate_branching_angle(*self*)

```
Calcualte the confluence branching angle (https://www.pnas.org/doi/10.1073/pnas.
↪1215218109)

Returns:
    float: The branching angle in degree
```

tojson(*self*)

```
Convert a pyconfluence object to json

Returns:
    json str: A json string
```

pyflowline.classes.link.pycelllink : public object

```
The cell link class

Args:
    object (obj): None

Returns:
    pycelllink: A link object
```

Public Functions

__init__(*self*, *pCell_start_in*, *pCell_end_in*, *pEdge_link_in*)

```
Initilize a link object

Args:
    pCell_start_in (pycell): The starting cell object
    pCell_end_in (pycell): The ending cell object
    pEdge_link_in (pyedge): An edge object that links two cells
```

tojson(*self*)

```
Convert a cell link object to a json string

Returns:
    json str: A json string
```

CHAPTER

FIFTEEN

ADDENDUM

15.1 Glossary

15.1.1 Structured mesh

In PyFlowline, structured mesh refers to meshes that have a repeating pattern or structure.

The following meshes are considered as structured:

1. Projected raster meshes (e.g. 100m by 100m)
2. GCS-based rectangle meshes (e.g. 0.5 degree by 0.5 degree)
3. Hexagon meshes (e.g. 100m by edge)
4. DGGS meshes (e.g., DGGrid meshes)

15.1.2 Unstructured mesh

In PyFlowline, unstructured mesh refers to meshes that don't have a repeating pattern or structure and the cell size varies from cell to cell.

The following meshes are considered as unstructured:

1. Model for Prediction Across Scales (MPAS) meshes
2. Triangulated irregular network (TIN) meshes

15.1.3 Great circle

In mathematics, a great circle or orthodrome is the circular intersection of a sphere and a plane passing through the sphere's center point.

15.1.4 DGGS

A discrete global grid (DGG) is a mosaic that covers the entire Earth's surface. Mathematically it is a space partitioning: it consists of a set of non-empty regions that form a partition of the Earth's surface. In a usual grid-modeling strategy, to simplify position calculations, each region is represented by a point, abstracting the grid as a set of region-points. Each region or region-point in the grid is called a cell.

15.1.5 TIN

In computer graphics, a triangulated irregular network (TIN) is a representation of a continuous surface consisting entirely of triangular facets (a triangle mesh), used mainly as Discrete Global Grid in primary elevation modeling.

15.1.6 MPAS

Model for Prediction Across Scales.

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

__eq__()
    built-in function, 41, 44, 46
__init__()
    built-in function, 41, 42, 45, 47, 48, 50, 51, 53, 55, 58
__ne__()
    built-in function, 42, 44, 46

B

basin_analyze()
    built-in function, 53
basin_build_confluence()
    built-in function, 53
basin_calculate_confluence_branching_angle()
    built-in function, 54
basin_calculate_flowline_length()
    built-in function, 54
basin_calculate_river_sinuosity()
    built-in function, 54
basin_convert_flowline_to_geojson()
    built-in function, 54
basin_evaluate()
    built-in function, 55
basin_evaluate_area_of_difference()
    built-in function, 55
basin_export()
    built-in function, 53
basin_export_basin_info_to_json()
    built-in function, 54
basin_export_config_to_json()
    built-in function, 54
basin_export_confluence_info_to_json()
    built-in function, 54
basin_export_flowline()
    built-in function, 53
basin_export_flowline_info_to_json()
    built-in function, 54
basin_flowline_simplification()
    built-in function, 53
basin_reconstruct_topological_relationship()
    built-in function, 53
built-in function
    __eq__(), 41, 44, 46
    __init__(), 41, 42, 45, 47, 48, 50, 51, 53, 55, 58
    __ne__(), 42, 44, 46
    basin_analyze(), 53
    basin_build_confluence(), 53
    basin_calculate_confluence_branching_angle(), 54
    basin_calculate_flowline_length(), 54
    basin_calculate_river_sinuosity(), 54
    basin_convert_flowline_to_geojson(), 54
    basin_evaluate(), 55
    basin_evaluate_area_of_difference(), 55
    basin_export(), 53
    basin_export_basin_info_to_json(), 54
    basin_export_config_to_json(), 54
    basin_export_confluence_info_to_json(), 54
    basin_export_flowline(), 53
    basin_export_flowline_info_to_json(), 54
    basin_flowline_simplification(), 53
    basin_reconstruct_topological_relationship(), 53
    calculate_cell_area(), 47, 49, 50, 52
    calculate_distance(), 42
    calculate_edge_length(), 47, 49, 50, 52
    calculate_flowline_sinuosity(), 46
    calculate_length(), 42, 45
    check_downstream(), 43, 45
    check_shared_vertex(), 43
    check_upstream(), 43, 45
    check_vertex_on_edge(), 44
    has_this_edge(), 47, 48, 50, 51
    is_overlap(), 43
    merge_upstream(), 45
    pyflowline.classes.confluence.pyconfluence.__init__(), 57
    pyflowline.classes.confluence.pyconfluence.calculate_b 57
    pyflowline.classes.confluence.pyconfluence.tojson(), 57
    pyflowline_analyze(), 56

pyflowline_evaluate(), 56
pyflowline_export(), 56
pyflowline_export_basin_config_to_json(), 57
pyflowline_export_config_to_json(), 57
pyflowline_export_mesh_info_to_json(), 56
pyflowline_merge_cell_info(), 56
pyflowline_mesh_generation(), 55
pyflowline_print(), 57
pyflowline_reconstruct_topological_relationship(), 55
pyflowline_run(), 56
pyflowline_setup(), 56
reverse(), 43, 45
share_edge(), 48–50, 52
split_by_length(), 43, 46
tojson(), 42, 44, 46, 48, 49, 51, 52, 54, 56, 58
toNvector(), 41
which_edge_cross_this_vertex(), 47, 49–51

C

calculate_cell_area()
built-in function, 47, 49, 50, 52
calculate_distance()
built-in function, 42
calculate_edge_length()
built-in function, 47, 49, 50, 52
calculate_flowline_sinuosity()
built-in function, 46
calculate_length()
built-in function, 42, 45
check_downstream()
built-in function, 43, 45
check_shared_vertex()
built-in function, 43
check_upstream()
built-in function, 43, 45
check_vertex_on_edge()
built-in function, 44

H

has_this_edge()
built-in function, 47, 48, 50, 51

I

is_overlap()
built-in function, 43

M

merge_upstream()
built-in function, 45

P

pyflowline.classes.confluence.pyconfluence
(*built-in class*), 57
pyflowline.classes.confluence.pyconfluence.__init__()
built-in function, 57
pyflowline.classes.confluence.pyconfluence.calculate_branc
built-in function, 57
pyflowline.classes.confluence.pyconfluence.tojson()
built-in function, 57
pyflowline_analyze()
built-in function, 56
pyflowline_evaluate()
built-in function, 56
pyflowline_export()
built-in function, 56
pyflowline_export_basin_config_to_json()
built-in function, 57
pyflowline_export_config_to_json()
built-in function, 57
pyflowline_export_mesh_info_to_json()
built-in function, 56
pyflowline_merge_cell_info()
built-in function, 56
pyflowline_mesh_generation()
built-in function, 55
pyflowline_print()
built-in function, 57
pyflowline_reconstruct_topological_relationship()
built-in function, 55
pyflowline_run()
built-in function, 56
pyflowline_setup()
built-in function, 56

R

reverse()
built-in function, 43, 45

S

share_edge()
built-in function, 48–50, 52
split_by_length()
built-in function, 43, 46

T

tojson()
built-in function, 42, 44, 46, 48, 49, 51, 52, 54, 56, 58
toNvector()
built-in function, 41

W

which_edge_cross_this_vertex()
built-in function, 47, 49–51