pyBlMstab Documentation

Release 0.1.5

Exneyder A. Montoya-Araque
Ludger O. Suarez-Burgoa

Oct 16, 2019

Contents:

`pybimstab` is an application software in **Python 3** to evaluate the factor of safety against sliding of slopes made of Blocks-In-Matrix (BIM) materials.

The assessment is donde by using the limit equilibrium method through the General Limit Equilibrium (GLE) method of Fredlund & Krahn (1977).

The slip surface has a tortuous geometry and is optimally found by using the math:*mathrm{A}^*ast* algorithm proposed by Hart, Nilsson & Raphael (1968).

The following plots are the final outcome of two different analysis:

Homogeneus slope

Slope made of BIM material

CHAPTER 1

Installation

1.1 Stable release

To install pyBIMstab, run this command in your terminal:

```
$ pip install pybimstab
```

This is the preferred method to install pyBIMstab, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.2 From sources

The sources for pyBIMstab can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/eamontoyaa/pybimstab
```

Or download the tarball:

```
$ curl  -OL https://github.com/eamontoyaa/pybimstab/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 2

Modules

This application software is divided into ten modules. The following sections correspond to the documentation of each module that defines the structure of the aplication software called `pyBIMstab`.

Each module contains the respective explanation of the classes and methods, sometimes supported by bibliographic references. The atttributes and input parameters are described and followed by examples for showing the different cases of application.

2.1 astar

Module for defining the classes related to the A* algorithm.

A* algorithm (pronounced as "A star") is a search algorithm proposed in 1968 by Hart, Nilsson & Raphael (1968). It optimally finds the minimum cost path in a graph from a start node s to a goal node g; in other words, the shortest path if the cost is given by its length. In addition, the algorithm does not expand as many nodes as other algorithms do; then, for a graph with a huge quantity of nodes, the computational performance is higher with respect to other search algorithms

The A* algorithm works constantly evaluating an evaluation function $f(n)$ composed of two parts: one is the actual cost of the optimum path traveled from s to the current node n given by the expression $g(n)$, and the second is the cost of the optimum path from the current node n to g given by the expression $h(n)$, which is the heuristic component of the algorithm and it could be for example either the euclidean or manhattan distance. Thereby, the evaluation function to measure the path cost is $f(n) = g(n) + h(n)$.

class astar.**PreferredPath**(*coordsIdx*, *factor*)

Bases: object

Creates an instance of an object that stores the indexes of a polyline in the space of the grid-graph that represents a preferential path to be followed when the A* algorithm is applied.

```
PreferredPath(coordsIdx, factor)
```

The object has an attribute called factor that represents a coefficient k that multiplies the distance d between the current node n and the polyline. Considering the above, the function for evaluating the total cost of a node is

modified as $f(n) = g(n) + h(n) + kd$

polyline
(2, n) array with the indexes of a polyline in the space of a grid-graph where the A* algorithm is applied. The first row corresponds to the rows and the second to the columns of the grid-graph.

Type *numpy.ndarray*

factor
Multiplier of the shortest distance between the current node and the polyline.

Type *int* or *float*

Examples

```
>>> from numpy import array
>>> from pybimstab.astar import PreferredPath
>>> coordsIdx = array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                       13, 13, 13, 13, 13, 13, 13, 13, 13, 13],
                      [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                       2, 3, 4, 5, 6, 7, 8, 9]])
>>> preferredPath = PreferredPath(coordsIdx, factor=1)
>>> preferredPath.__dict__
{'coordsIdx': array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 13,
                     13, 13, 13, 13, 13, 13, 13, 13, 13],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                     2, 3, 4, 5, 6, 7, 8, 9]]),
 'factor': 1}
```

class astar.**Node**(*pos*, *father=None*, *gCost=None*, *hCost=None*, *val=0*)
Bases: object

Creates an instance of an object that defines the structure of a node that belongs to a grid-graph where is wanted to find the optimum path through the A* algorithm.

```
Node(pos=None, father=None, gCost=None, hCost=None, val=0)
```

pos
Indexes of the the current node in the grid-graph.

Type *tuple*

father
Indexes of the father node of the current node. Default value is None.

Type *tuple*

gCost
Length of the traveled path from the start node to the current node. Default value is None.

Type *int* or *float*

hCost
Heuristic length of the path from the current node to the goal node. Default value is None.

Type *int* or *float*

val
Value that store the node cell in the matrix that defines the grid-graph. Default value is 0.

Type *int*

Note: The class `Node` requires numpy and shapely

Examples

```
>>> node = Node(pos=(6, 7), father=(5, 7), gCost=5, hCost=10, val=1)
>>> node.__dict__
{'father': (5, 7), 'gCost': 5, 'hCost': 10, 'pos': (6, 7), 'val': 1}
```

getHcost (*goalNode*, *heuristic='manhattan'*, *preferredPath=None*)

Method to obtain the heuristic component, $h(n)$, that estimates the cost (or length) of the shortest path from the current node n to the goal node.

It must be selected either `manhattan` or `euclidean` as the model to estimate the length of the optimum path.

- **manhattan** is the sum of the cathetus of the right triangle defined by the current node and the goal node.
- **euclidean** is the length of the hypotenuse of the right triangle defined by the current node and the goal node.

It is possible to append a polyline to force the path to follow a preferential path.

Parameters

- **goalNode** (*Node* object) – object with the structure of the goal node.
- **heuristic** (*str*) – Name of the geometric model to determine the heuristic distance. It must be selected either `manhattan` or `euclidean`. The first one is the default value.
- **preferredPath** (*Node* object) – Optional argument of the class `Node` to force the path. `None` is the default value.

Returns value of the estimated heuristic distance of the opmtimum path.

Return type (*int* or *float*)

Examples

```
>>> from pybimstab.astar import Node
>>> goalNode = Node(pos=(9, 9))
>>> node = Node(pos=(0, 0))
>>> node.getHcost(goalNode, heuristic='manhattan',
>>>               preferredPath=None)
18
>>> from pybimstab.astar import Node
>>> goalNode = Node(pos=(9, 9))
>>> node = Node(pos=(0, 0))
>>> node.getHcost(goalNode, heuristic='euclidean',
>>>               preferredPath=None)
12.727922061357855
```

getGcost (*fatherNode*)

Method to obtain the cumulated cost $g(n)$, of the traveled path from the start node to the current node n.

Parameters **fatherNode** (`Node` object) – object with the structure of the current node's father.

Returns traveled-path length from the the start node to the current node.

Return type (*int* or *float*)

Examples

```
>>> from pybimstab.astar import Node
>>> node = Node(pos=(9, 9))
>>> fatherNode = Node(pos=(9, 8), gCost=15)
>>> node.getGcost(fatherNode)
16
```

```
>>> from pybimstab.astar import Node
>>> node = Node(pos=(9, 9))
>>> fatherNode = Node(pos=(8, 8), gCost=15)
>>> node.getGcost(fatherNode)
16.4142
```

class astar.**Astar**(*grid*, *startNode*, *goalNode*, *heuristic='manhattan'*, *reverseLeft=True*, *reverseUp=True*, *preferredPath=None*)

Bases: object

Creates an instance of an object that defines the optimum path into a grid-graph maze, from a start node to a goal node given.

```
Astar(grid, startNode, goalNode, heuristic='manhattan',
      reverseLeft=True, reverseUp=True, preferredPath=None)
```

gridGraph
object with the structure of maze where is wanted to find the optimum path.

Type *MazeStructure* object

startNode
indexes of the `gridGraph.matrix` where the initial node is located. It has to be a matrix cell, *ie*, `gridGraph.matrix[startNode]==0`.

Type *tuple*, *list* or *numpy.ndarray*

goalNode
indexes of the `gridGraph.matrix` where the ending node is located. It has to be a matrix cell, *ie*, `gridGraph.matrix[goalNode]==0`.

Type *tuple*, *list* or *numpy.ndarray*

heuristic
Name of the geometric model to determine the heuristic distance. It must be selected either `manhattan` or `euclidean`. The first one is the default value.

Type *str*

reverseLeft
Logical variable to allow or not reverses movements to the left. Default value is `True`.

Type *bool*

reverseUp
Logical variable to allow or not reverses movements to upward. Default value is `True`.

Type *bool*

***forcedPath**
Optional aguments to force the optimum path close to a specific polyline.

Type *PreferredPath* object

Note: The class `Astar` requires NumPy and Matplotlib.

Examples

```
>>> from numpy import array
>>> from pybimstab.astar import PreferredPath, Astar
>>> grid = array([[ 0,  0,  0,  0,  1,  0,  0,  0,  0,  0],
>>>               [ 0,  0,  0,  0,  1,  1,  0,  0,  0,  0],
>>>               [ 0,  0,  0,  1,  1,  0,  1,  0,  1,  0],
>>>               [ 0,  1,  1,  0,  0,  0,  0,  1,  0,  0],
>>>               [ 0,  0,  0,  1,  0,  0,  0,  1,  1,  1],
>>>               [ 0,  0,  1,  0,  1,  1,  0,  0,  0,  0],
>>>               [ 0,  0,  1,  0,  1,  0,  1,  0,  0,  0],
>>>               [ 1,  1,  0,  1,  0,  0,  0,  0,  1,  1],
>>>               [ 1,  0,  0,  0,  0,  0,  0,  0,  0,  0],
>>>               [ 0,  0,  0,  0,  1,  0,  1,  0,  0,  1],
>>>               [ 0,  1,  0,  0,  0,  1,  0,  0,  0,  0],
>>>               [ 1,  0,  0,  0,  0,  0,  0,  0,  0,  0],
>>>               [ 0,  0,  0,  0,  0,  0,  0,  0,  1,  0],
>>>               [ 1,  0,  0,  0,  1,  0,  0,  0,  0,  0]
>>>               ])
>>> coordsIdx = array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
>>>                     13, 13, 13, 13, 13, 13, 13, 13, 13, 13],
>>>                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
>>>                     2, 3, 4, 5, 6, 7, 8, 9]])
>>> preferredPath = PreferredPath(coordsIdx, factor=1)
>>> astar = Astar(grid, startNode=(0, 0), goalNode=(13, 9),
>>>               heuristic='manhattan', reverseLeft=True,
>>>               reverseUp=True, preferredPath=preferredPath)
>>> astar.__dict__.keys()
dict_keys(['grid', 'startNode', 'goalNode', 'heuristic', 'reverseLeft',
           'reverseUp', 'preferredPath', 'mazeStr', 'optimumPath'])
```

defineMazeStr()
Defines the clean structure of the grid-graph using objects instaced from the class `Node` for each node (or cell of the grid).

Those cells with value equal to one or minus one will have a g-value equal to infinite because over those cells, a path is impossible to be traced.

Returns array with the same shape of the input grid where each cell is an object from the class `Node` with the structure of its respective node.

Return type (*numpy.ndarray*)

Examples

```
>>> # after executing the example of the class
>>> astar.mazeStr[0, 0].__dict__
```

(continues on next page)

(continued from previous page)

```
{'father': (0, 0), 'gCost': 0, 'hCost': 22.0, 'pos': (0, 0),
 'val': 0}
```

getNeighbours(*node*)

Method for obtaining the possible neighbours of an specific node that belongs to the grid given.

Each neighbour is given as a tuple with the indexes of the grid.

Parameters **node** (Node object) – object with the structure of the node which is wanted to know its possible neighbours.

Returns Tuples with the indexes of possible neighbours of the node in question.

Return type (*list*)

Examples

```
>>> # after executing the example of the class
>>> from pybimstab.astar import Node
>>> astar.getNeighbours(Node(pos=(1, 1)))
[(0, 1), (0, 2), (1, 2), (2, 2), (2, 1), (2, 0), (1, 0), (0, 0)]
>>> astar.getNeighbours(Node(pos=(0, 0)))
[(0, 1), (1, 1), (1, 0)]
>>> astar.getNeighbours(Node(pos=(0, 1)))
[(0, 2), (1, 2), (1, 1), (1, 0), (0, 0)]
>>> astar.getNeighbours(Node(pos=(0, 2)))
[(1, 2), (1, 1), (0, 1)]
>>> astar.getNeighbours(Node(pos=(1, 2)))
[(0, 2), (2, 2), (2, 1), (1, 1), (0, 1)]
>>> astar.getNeighbours(Node(pos=(2, 2)))
[(1, 2), (2, 1), (1, 1)]
>>> astar.getNeighbours(Node(pos=(2, 1)))
[(1, 1), (1, 2), (2, 2), (2, 0), (1, 0)]
>>> astar.getNeighbours(Node(pos=(2, 0)))
[(1, 0), (1, 1), (2, 1)]
>>> astar.getNeighbours(Node(pos=(1, 0)))
[(0, 0), (0, 1), (1, 1), (2, 1), (2, 0)]
```

getWayBack(*node*)

Method for obtaining the whole way back of an specific node which has been opened by the *A star* algorithm.

Parameters **node** (Node object) – object with the structure of the node which is wanted to know its way back.

Returns $(2 \times n)$ array where n is the number of nodes where the path has crossed; the first row of the array contains the abscisses and the second one contains the ordinates of the nodes into the grid-graph

Return type (*numpy.ndarray*)

Examples

```
>>> import numpy as np
>>> from pybimstab.astar import PreferredPath, Node, Astar
```

(continues on next page)

(continued from previous page)

```
>>> grid = np.zeros((3,3))
>>> grid[1:3, 0:2] = 1
>>> astar = Astar(grid, startNode=(0, 0), goalNode=(2, 2),
>>>               heuristic='manhattan', reverseLeft=True,
>>>               reverseUp=True, preferredPath=None)
>>> # returning the way back
>>> astar.getWayBack(astar.mazeStr[2, 2])
array([[2, 1, 0, 0],
       [2, 1, 1, 0]])
>>> astar.getWayBack(astar.mazeStr[1, 2])
array([[1, 0, 0],
       [2, 1, 0]])
>>> astar.getWayBack(astar.mazeStr[1, 1])
ValueError: Input node is a block. It doesn't have a wayback
```

getPath()

Method for obtaining the optimum path between two points into a grid-graph through the A* algorithm Hart, Nilsson & Raphael (1968).

Returns $(2 \times n)$ array where n is the number of nodes where the path has crossed; the first row of the array contains the abscisses and the second one contains the ordinates of the nodes into the grid-graph

Return type (*numpy.ndarray*)

Examples

```
>>> from numpy import array
>>> from pybimstab.astar import PreferredPath, Astar
>>> grid = array([[ 0,  0,  0,  0,  1,  0,  0,  0,  0,  0],
>>>               [ 0,  0,  0,  0,  1,  1,  0,  0,  0,  0],
>>>               [ 0,  0,  0,  1,  1,  0,  1,  0,  1,  0],
>>>               [ 0,  1,  1,  0,  0,  0,  0,  1,  0,  0],
>>>               [ 0,  0,  0,  1,  0,  0,  0,  1,  1,  1],
>>>               [ 0,  0,  1,  0,  1,  1,  0,  0,  0,  0],
>>>               [ 0,  0,  1,  0,  1,  0,  1,  0,  0,  0],
>>>               [ 1,  1,  0,  1,  0,  0,  0,  0,  1,  1],
>>>               [ 1,  0,  0,  0,  0,  0,  0,  0,  0,  0],
>>>               [ 0,  0,  0,  0,  1,  0,  1,  0,  0,  1],
>>>               [ 0,  1,  0,  0,  0,  1,  0,  0,  0,  0],
>>>               [ 1,  0,  0,  0,  0,  0,  0,  0,  0,  0],
>>>               [ 0,  0,  0,  0,  0,  0,  0,  0,  1,  0],
>>>               [ 1,  0,  0,  0,  1,  0,  0,  0,  0,  0]
>>>               ])
>>> coordsIdx = array(
>>>     [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 13, 13, 13,
>>>       13, 13, 13, 13, 13, 13, 13],
>>>      [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 3, 4,
>>>       5, 6, 7, 8, 9]])
>>> preferredPath = PreferredPath(coordsIdx, factor=1)
```

```
>>> # without a forced path
>>> astar = Astar(grid, startNode=(0, 0), goalNode=(13, 9),
>>>               heuristic='manhattan', reverseLeft=True,
>>>               reverseUp=True, preferredPath=None)
```

(continues on next page)

(continued from previous page)

```
>>> astar.getPath()
array(
    [[13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  4,  3,  2,  1,  0],
     [ 9,  9,  9,  9,  8,  8,  7,  7,  6,  5,  4,  3,  2,  1,  0]])
```

```
>>> # with a forced path
>>> astar = Astar(grid, startNode=(0, 0), goalNode=(13, 9),
>>>               heuristic='manhattan', reverseLeft=True,
>>>               reverseUp=True, preferredPath=preferredPath)
>>> astar.getPath()
array([
    [13, 13, 13, 13, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0],
    [ 9, 8, 7, 6, 5, 4,  3, 2, 2, 2, 2, 1, 0, 0, 0, 0, 0, 0]])
```

```
>>> from numpy import array
>>> from pybimstab.astar import PreferredPath, Astar
>>> grid = array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 1, 0, 0, 1, 1, 0, 0],
                  [1, 1, 0, 0, 1, 0, 1, 0, 0, 0],
                  [0, 0, 0, 1, 1, 0, 1, 0, 0, 1],
                  [0, 0, 1, 1, 0, 1, 1, 1, 0, 0],
                  [0, 0, 0, 1, 1, 0, 1, 0, 0, 1],
                  [0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
                  [0, 0, 0, 1, 1, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
                  [0, 0, 1, 0, 0, 1, 0, 0, 0, 0]])
>>> astar = Astar(grid, startNode=(0, 0), goalNode=(9, 9),
>>>               heuristic='manhattan', reverseLeft=True,
>>>               reverseUp=True, preferredPath=None)
>>> astar.getPath()
array([[9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [9, 8, 8, 8, 8, 8, 8, 8, 8, 7, 6, 5, 4, 3, 2, 1, 0]])
>>> astar = Astar(grid, startNode=(0, 0), goalNode=(9, 9),
>>>               heuristic='euclidean', reverseLeft=True,
>>>               reverseUp=True, preferredPath=None)
>>> astar.getPath()
array([[9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 0, 0, 0],
       [9, 8, 7, 6, 5, 4, 5, 5, 4, 3, 2, 1, 0]])
```

plot (*plotPreferredPath=False*)

Method for generating a graphic of the optimum path returned from the `astar` method.

Parameters **plotForcedPath** (*bool*) – logical variable to check if the forced path exists and is wanted to plot.

Returns object with the matplotlib structure of the plot. You might use it to save the figure for example.

Return type (*matplotlib.figure.Figure*)

Examples

```
>>> from numpy import array
>>> from pybimstab.astar import PreferredPath, Astar
>>> grid = array([[0,  0,  0,  0,  1,  0,  0,  0,  0,  0],
```

(continues on next page)

(continued from previous page)

```
>>>                [0,  0,  0,  0,  1,  1,  0,  0,  0,  0],
>>>                [0,  0,  0,  1,  1,  0,  1,  0,  1,  0],
>>>                [0,  1,  1,  0,  0,  0,  0,  1,  0,  0],
>>>                [0,  0,  0,  1,  0,  0,  0,  1,  1,  1],
>>>                [0,  0,  1,  0,  1,  1,  0,  0,  0,  0],
>>>                [0,  0,  1,  0,  1,  0,  1,  0,  0,  0],
>>>                [1,  1,  0,  1,  0,  0,  0,  0,  1,  1],
>>>                [1,  0,  0,  0,  0,  0,  0,  0,  0,  0],
>>>                [0,  0,  0,  0,  1,  0,  1,  0,  0,  1],
>>>                [0,  1,  0,  0,  0,  1,  0,  0,  0,  0],
>>>                [1,  0,  0,  0,  0,  0,  0,  0,  0,  0],
>>>                [0,  0,  0,  0,  0,  0,  0,  0,  1,  0],
>>>                [1,  0,  0,  0,  1,  0,  0,  0,  0,  0]
>>>                ])
>>> coordsIdx = array(
>>>     [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 13, 13, 13,
>>>       13, 13, 13, 13, 13, 13, 13],
>>>      [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 3, 4,
>>>       5, 6, 7, 8, 9]])
>>> preferredPath = PreferredPath(coordsIdx, factor=1)
>>> for typePP in [None, preferredPath]:
>>>     astar = Astar(grid, startNode=(0, 0), goalNode=(13, 9),
>>>                   heuristic='manhattan', reverseLeft=True,
>>>                   reverseUp=True, preferredPath=typePP)
>>>     fig = astar.plot(plotPreferredPath=True)
```

```
>>> from numpy import array
>>> from pybimstab.astar import Astar
>>> grid = array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 1, 0, 0, 1, 1, 0, 0],
                  [1, 1, 0, 0, 1, 0, 1, 0, 0, 0],
                  [0, 0, 0, 1, 1, 0, 1, 0, 0, 1],
                  [0, 0, 1, 1, 0, 1, 1, 1, 0, 0],
                  [0, 0, 0, 1, 1, 0, 1, 0, 0, 1],
                  [0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
                  [0, 0, 0, 1, 1, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
                  [0, 0, 1, 0, 0, 1, 0, 0, 0, 0]])
>>> for heuristic in ['manhattan', 'euclidean']:
>>>     astar = Astar(grid, startNode=(0, 0), goalNode=(9, 9),
>>>                   heuristic=heuristic, reverseLeft=True,
>>>                   reverseUp=True, preferredPath=None)
>>>     fig = astar.plot()
```

```
>>> from numpy import array
>>> from pybimstab.bim import BlocksInMatrix
>>> from pybimstab.astar import Astar
>>> seed = 111  # for repeatibilty
>>> boundary = array([[-5, 0, 5, 0, -5], [0, 10, 0, -10, 0]])
>>> bim = BlocksInMatrix(slopeCoords=boundary, blockProp=0.2,
>>>                      tileSize=0.4, seed=seed)
```

(continues on next page)

(continued from previous page)

```
>>> astar = Astar(bim.grid, startNode=(0, 12), goalNode=(49, 12),
>>>               heuristic='manhattan', reverseLeft=True,
>>>               reverseUp=True, preferredPath=None)
>>> fig = astar.plot()
```

2.2 bim

Module for defining the class related the structure of the BIM.

class bim.**BlocksInMatrix**(*slopeCoords*, *blockProp*, *tileSize*, *seed=None*)

Bases: object

Creates an instance of an object that defines the structure of the block-in-rock material (BIM) that composes the slope.

```
BlocksInMatrix(slopeCoords, blockProp, tileSize, seed=None)
```

The BIM is defined as a grided array of vertical square tiles. Each tile is composed either of block or matrix. Those cells that correspond to a block-tile have value one (1), those cells that correspond to a matrix-tile have value zero (0), and those cells tath are located outside the polygon have vale minus one (-1).

slopeCoords
Coordinates of the polygon within which the BIM is defined. It is expected that the polygon corresponds to the slope boundary that is obtained with the method `defineBoundary` either from the classes `AnthropicSlope` or `NaturalSlope` (module `slope`), however it works for any closed polygon.

Type (n, 2) *numpy.ndarray*

blockProp
Proportion of blocks relative to the total volume of the BIM. It's given as a value between 0 and 1.

Type *float*

tileSize
Length of each tile-side.

Type `int` or `float`

seed
Seed value for repeatability in the random generation of the blocks.

Type `float`

Note: The class `BlocksInMatrix` requires numpy and matplotlib.

Examples

```
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
```

(continues on next page)

(continued from previous page)

```
>>>                         crownDist=10, toeDist=10)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.25,
>>>                      tileSize=0.25)
>>> bim.__dict__.keys()
dict_keys(['slopeCoords', 'blockProp', 'tileSize', 'seed', 'grid',
           'xCells', 'yCells'])
```

defineGrid()

Method to create the grid-structure of the BIM into de boundary.

Those cells that correspond to a block-tile have value one (1), those cells that correspond to a matrix-tile have value zero (0), and those cells tath are located outside the polygon have vale minus one (-1).

Returns $(m \times n)$ matrix that defines a grid-graph that represents the structure of the BIM.

Return type (*numpy.ndarray*)

Examples

```
>>> from numpy import array
>>> from pybimstab.bim import BlocksInMatrix
>>> slopeCoords = array([[0, 1, 1, 0, 0], [0, 0, 1, 1, 0]])
>>> bim = BlocksInMatrix(slopeCoords=slopeCoords, blockProp=0.5,
>>>                      tileSize=0.1, seed=123)
>>> bim.defineGrid()
array([[1., 0., 0., 1., 1., 0., 1., 1., 0., 0.],
       [0., 1., 0., 0., 0., 1., 0., 0., 1., 1.],
       [1., 1., 1., 1., 1., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0., 0., 0., 1., 1., 1.],
       [1., 0., 0., 0., 1., 0., 0., 1., 1., 1.],
       [0., 1., 1., 1., 0., 0., 0., 1., 1., 1.],
       [1., 1., 1., 1., 1., 0., 1., 0., 0., 1.],
       [0., 1., 1., 1., 0., 1., 1., 0., 0., 1.],
       [0., 1., 1., 0., 1., 1., 0., 0., 0., 0.]])
```

```
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
>>>                        crownDist=10, toeDist=10)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.25,
>>>                      tileSize=0.25, seed=123)
>>> bim.defineGrid()
array([[-1., -1., -1., ..., -1., -1., -1.],
       [ 0.,  1.,  0., ...,  1.,  0.,  1.],
       [ 1.,  0.,  0., ...,  1.,  0.,  0.],
       ...,
       [ 0.,  1.,  0., ..., -1., -1., -1.],
       [ 0.,  0.,  0., ..., -1., -1., -1.],
       [ 0.,  0.,  0., ..., -1., -1., -1.]])
```

plot()

Method for generating a graphic of the grid structure of the BIM.

Returns object with the matplotlib structure of the plot. You might use it to save the figure for example.

Return type (*matplotlib.figure.Figure*)

Examples

```
>>> from numpy import array
>>> from pybimstab.bim import BlocksInMatrix
>>> slopeCoords = array([[0, 1, 1, 0, 0], [0, 0, 1, 1, 0]])
>>> bim = BlocksInMatrix(slopeCoords=slopeCoords, blockProp=0.5,
>>>                      tileSize=0.1, seed=123)
>>> fig = bim.plot()
```

```
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
>>>                        crownDist=10, toeDist=10)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.2,
>>>                      tileSize=0.25, seed=123)
>>> fig = bim.plot()
```

```
>>> from numpy import array
>>> from pybimstab.slope import NaturalSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> terrainCoords = array(
>>>     [[-2.49, 0.1 , 1.7 , 3.89, 5.9 , 8.12, 9.87, 13.29, 20.29,
>>>       21.43, 22.28, 23.48, 24.65, 25.17],
>>>      [18.16, 17.88, 17.28, 15.73, 14.31, 13.58, 13, 3.61, 3.61,
>>>       3.32, 2.71, 2.23, 1.21, 0.25]])
>>> slope = NaturalSlope(terrainCoords)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.25,
>>>                      tileSize=0.4, seed=123)
>>> fig = bim.plot()
```

2.3 polygon

Module for defining the class related to a Polygon and to verify what points are inside it.

class polygon.**Polygon**(*coordinates*)

Bases: object

Creates an instance of an object that defines a two dimension polygon.

```
Polygon(coordinates)
```

coordinates

Coordinates of vertices of the polygon.

Type (n, 2) *numpy.ndarray*

Note: The class *Polygon* requires numpy and matplotlib.

Examples

```
>>> from numpy import array
>>> from pybimstab.polygon import Polygon
>>> coords = array([[0. , 1. , 1. , 0. ], [0. , 0. , 1.5, 1.5]])
>>> polygon = Polygon(coordinates=coords)
>>> polygon.__dict__
{'coordinates': array([[0. , 1. , 1. , 0. ],
                       [0. , 0. , 1.5, 1.5]])}
```

isinside(*x*, *y*, *meshgrid=False*, *want2plot=False*)

Method to know if the point(s) (x, y) is inside of the instanced polygon.

`x` and `y` could be iterable structures, even, they could define a meshgrid.

Parameters

- **x** (*int*, *float* or (n,) *numpy.ndarray*) – abscissa of the point(s) to check if is/are inside the polygon.
- **y** (*int*, *float* or (n,) *numpy.ndarray*) – ordinate of the point(s) to check if is/are inside the polygon.
- **meshgrid** (*bool*) – variable to creck if `x` and `y` define a grid. The default value is `False`.
- **want2plot** (*bool*) – variable to creck if a plot is wanted. The default value is `False`.

Returns Array of boolean values where *True* means that the point is inside and *False* to points that is outside of the polygon.

Return type *numpy.ndarray*

Note: The method `isinside` does not work properly for points on the boundaries of the polygon.

Examples

```
>>> from numpy import array
>>> from pybimstab.polygon import Polygon
>>> coords = array([[0. , 1. , 1. , 0. ], [0. , 0. , 1.5, 1.5]])
>>> x, y = 0.5, 2
>>> polygon = Polygon(coordinates=coords)
>>> polygon.isinside(x=x, y=y, meshgrid=False, want2plot=True)
array([True], dtype=bool)
```

```
>>> from numpy import array
>>> from pybimstab.polygon import Polygon
>>> coords = array([[0, 1, 1, 0.], [0, 0, 1.5, 1.5]])
>>> x = array([0.3, 0.5, 0.7, 1.2, 1.0])
>>> y = array([0.6, 1., 1.4, 2.4, 0])
>>> polygon = Polygon(coordinates=coords)
>>> polygon.isinside(x=x, y=y, meshgrid=False, want2plot=True)
array([True, True, True, False, False], dtype=bool)
```

```
>>> from numpy import array
>>> from pybimstab.polygon import Polygon
>>> coords = array([[0, 1, 1, 0.], [0, 0, 1.5, 1.5]])
>>> x = array([0.3, 0.5, 0.7, 1.2, 1.0])
>>> y = array([0.6, 1., 1.4, 2.4, 0])
>>> polygon = Polygon(coordinates=coords)
>>> polygon.isinside(x=x, y=y, meshgrid=True, want2plot=True)
array([[ True,  True,  True, False,  True],
       [ True,  True,  True, False,  True],
       [ True,  True,  True, False,  True],
       [False, False, False, False, False],
       [False, False, False, False, False]], dtype=bool)
```

```
>>> import numpy as np
>>> from pybimstab.polygon import Polygon
>>> xC = [np.cos(theta) for theta in np.linspace(0, 2*np.pi, 100)]
>>> yC = [np.sin(theta) for theta in np.linspace(0, 2*np.pi, 100)]
>>> coords = np.array([xC, yC])
>>> np.random.seed(123)
>>> x = np.random.uniform(-1, 1, 100)
>>> y = np.random.uniform(-1, 1, 100)
>>> polygon = Polygon(coordinates=coords)
>>> polygon.isinside(x=x, y=y, meshgrid=False, want2plot=True);
```

2.4 slices

Module for defining the class related to the slices and their structure.

class slices.**MaterialParameters**(*cohesion*, *frictAngle*, *unitWeight*, *blocksUnitWeight=None*, *wtUnitWeight=None*)

Bases: object

Creates an instance of an object that defines the structure of the material where the stabilibity analysis is performed.

```
MaterialParameters(cohesion, frictAngle, unitWeight,
                   blocksUnitWeight=None, wtUnitWeight=None)
```

Contains the strength parameters (Mohr-Coulomb failure criterion) and the different unit weigths.

cohesion
Intercept of the Mohr-Coulomb envelope.

Type *int* or *float*

frictAngle
Angle of the Mohr-Coulomb envelope.

Type *int* or *float*

unitWeight
Unit weight of the soil or the matrix in the case of a Blocks-in-Matrix material.

Type *int* or *float*

blocksUnitWeight
Unit weight of the blocks in the case of a Blocks-in-Matrix material. `None` is the default value that means there are no blocks.

Type *int* or *float*

wtUnitWeight
Unit weight of the water. `None` is the default value that means there are no water table, or it is located under the base of a specific slice.

Type *int* or *float*

Examples

```
>>> from pybimstab.slices import MaterialParameters
>>> material = MaterialParameters(
>>>     cohesion=15, frictAngle=23, unitWeight=17, blocksUnitWeight=21,
>>>     wtUnitWeight=9.8)
>>> material.__dict__
{'blocksUnitWeight': 21,
 'cohesion': 15,
 'frictAngle': 23,
 'mtxUnitWeight': 17,
 'wtUnitWeight': 9.8}
```

class slices.**SliceStr**(*material*, *terrainLS*, *slipSurfLS*, *watertabLS=None*, *bim=None*)
Bases: `object`

Creates an instance of an object that defines the structure of a slice of the soil mass above the slip surface.

```
SliceStr(material, terrainLS, slipSurfLS, watertabLS=None, bim=None)
```

The structure contains the required data for performing the slope stability assessment by the limit equilibrium method.

material
object with the parameters of the material that composes the slice.

Type *MaterialParameters* object

terrainLS
Top of the slice that coincides with the terrain surface.

Type *shapely.geometry.linestring.LineString*

slipSurfLS
Bottom of the slice that coincides with the slip surface.

Type *shapely.geometry.linestring.LineString*

watertabLS
Polyline that coincides with the water table. It could be located below or crossing the base of the slice, between the bottom and the top or absent. `None` is the default value.

Type *shapely.geometry.linestring.LineString*

bim
object with the structure of the slope made of the Blocks-in-Matrix material. `None` is the default value and means that there is not a BIM structure

Type *BimStructure* object

Note: The class `SliceStr` requires copy, numpy, matplotlib and shapely.

Examples

```
>>> from shapely.geometry import LineString
>>> from pybimstab.slices import MaterialParameters, SliceStr
>>> material = MaterialParameters(cohesion=15, frictAngle=23,
>>>                               unitWeight=17)
>>> terrainLS = LineString([(6, 8), (7, 6.5)])
>>> slipSurfLS = LineString([(6, 3.395), (7, 2.837)])
>>> watertabLS = LineString([(6, 5), (7, 4)])
>>> slice_ = SliceStr(material, terrainLS, slipSurfLS, watertabLS,
>>>                   bim=None)
>>> slice_.__dict__.keys()
dict_keys(['material', 'terrainLS', 'slipSurfLS', 'watertabLS', 'bim',
           'terrainCoords', 'slipSurfCoords', 'watertabCoords',
           'sliceLS', 'coords', 'xMin', 'xMax', 'yMin', 'yMax', 'area',
           'width', 'baseSlope', 'alpha', 'baseLength', 'l',
           'topLength', 'topSlope', 'topInclinatDeg', 'midHeight',
           'midWatTabHeight'])
```

defineStructure()

Method for defining the geometric structure of the slice including the variables required to perform the slope stability assessment by the limit equilibrium method.

Returns Coordinates of the slice boundary. First row contains the abscises and the second one contains the ordinates.

Return type ((2, n) *numpy.ndarray*)

Examples

```
>>> from shapely.geometry import LineString
>>> from pybimstab.slices import MaterialParameters, SliceStr
>>> material = MaterialParameters(cohesion=15, frictAngle=23,
>>>                               unitWeight=17)
>>> terrainLS = LineString([(6, 8), (7, 6.5)])
>>> slipSurfLS = LineString([(6, 3.395), (7, 2.837)])
>>> watertabLS = LineString([(6, 5), (7, 4)])
>>> slice_ = SliceStr(material, terrainLS, slipSurfLS, watertabLS,
>>>                   bim=None)
>>> slice_.defineStructure()
array([[6.   , 7.   , 7.   , 6.   , 6.   ],
       [8.   , 6.5  , 2.837, 3.395, 8.   ]])
```

extractBim()

Returns an object from the class `bim.BlocksInMatrix` where is stored the structure of the local Blocks-In-Matrix material inside the slice.

Examples

```
>>> from shapely.geometry import LineString
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> from pybimstab.slices import MaterialParameters, SliceStr
>>> material = MaterialParameters(cohesion=15, frictAngle=23,
>>>                               unitWeight=17)
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5, depth=2)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.2,
>>>                      tileSize=0.5, seed=123)
>>> terrainLS = LineString([(6, 8), (7, 6.5)])
>>> slipSurfLS = LineString([(6, 3.395), (7, 2.837)])
>>> watertabLS = LineString([(6, 5), (7, 4)])
>>> slice_ = SliceStr(material, terrainLS, slipSurfLS, watertabLS,
>>>                   bim=bim)
>>> localBIM = slice_.extractBim()
>>> localBIM.__dict__.keys()
dict_keys(['slopeCoords', 'blockProp', 'tileSize', 'seed', 'grid',
           'xCells', 'yCells'])
```

plot()
Method for plotting the slice.

Examples

```
>>> from shapely.geometry import LineString
>>> from pybimstab.slices import MaterialParameters, SliceStr
>>> material = MaterialParameters(cohesion=15, frictAngle=23,
>>>                               unitWeight=17)
>>> terrainLS = LineString([(6, 8), (7, 6.5)])
>>> slipSurfLS = LineString([(6, 3.395), (7, 2.837)])
>>> watertabLS = LineString([(6, 5), (7, 4)])
>>> slice_ = SliceStr(material, terrainLS, slipSurfLS, watertabLS,
>>>                   bim=None)
>>> fig = slice_.plot()
```

```
>>> from shapely.geometry import LineString
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> from pybimstab.slices import MaterialParameters, SliceStr
>>> material = MaterialParameters(cohesion=15, frictAngle=23,
>>>                               unitWeight=17)
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5, depth=2)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.2,
>>>                      tileSize=0.5, seed=123)
>>> terrainLS = LineString([(6, 8), (7, 6.5)])
>>> slipSurfLS = LineString([(6, 3.395), (7, 2.837)])
>>> watertabLS = LineString([(6, 5), (7, 4)])
>>> slice_ = SliceStr(material, terrainLS, slipSurfLS, watertabLS,
>>>                   bim=bim)
>>> fig = slice_.plot()
```

class slices.**Slices**(*material*, *slipSurfCoords*, *slopeCoords*, *numSlices=20*, *watertabCoords=None*, *bim=None*)

Bases: object

Creates an instance of an object that defines the structure of the slices that the soil mass above the slip surface is divided.

```
Slices(material, slipSurfCoords, slopeCoords, numSlices=20,
       watertabCoords=None, bim=None)
```

slipSurfCoords
Absolute coordinates of the vertices of the polyline which defines the slip surface. First row contains the abcsises and the second row contains the ordinates.

Type (2, n) *numpy.ndarray*

slopeCoords
Absolute coordinates of the vertices of the polyline which defines the slip surface. First column contains the abcsises and the second one contains the ordinates.

Type (n, 2) *numpy.ndarray*

material
object with the parameters of the material that composes the slope.

Type *MaterialParameters* object

numSlices
Number of slices in which the soil above the slip surface is going to be divided. 20 is the default value.

Type *int*

watertabCoords
Absolute coordinates of the vertices of the polyline which defines the water table. First row contains the abcsises and the second row contains the ordinates. It is like an optional argument and if there is not a water table, set it as None, which is the default value.

Type (2, n) *numpy.ndarray*

bim
object with the structure of the slope made of the BIM-material. It is like an optional argument and if there is not a BIM structure, set it as None, which is the default value.

Type *BimStructure* object

Note: The class Slices requires numpy, scipy, matplotlib and shapely.

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.slices import MaterialParameters, Slices
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
>>> surface = CircularSurface(slopeCoords=slope.coords,
```

(continues on next page)

(continued from previous page)

```
>>>                           dist1=2, dist2=10, radius=9)
>>> material = MaterialParameters(
>>>     cohesion=15, frictAngle=23, unitWeight=17,
>>>     blocksUnitWeight=21, wtUnitWeight=9.8)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=10,
>>>     watertabCoords=None, bim=None)
>>> slices.__dict__.keys()
dict_keys(['material', 'numSlices', 'slipSurfCoords', 'slopeCoords',
           'watertabCoords', 'bim', 'rotationPt', 'slices'])
```

fitCircle()

Method for adjusting a circumference to a cloud of points.

Returns Dictionary with the radius and coordinates of center.

Return type (*dict*)

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.slices import MaterialParameters, Slices
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=2, dist2=10, radius=9)
>>> material = MaterialParameters(
>>>     cohesion=15, frictAngle=23, unitWeight=17,
>>>     blocksUnitWeight=21, wtUnitWeight=9.8)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=10,
>>>     watertabCoords=None, bim=None)
>>> slices.fitCircle()
{'center': array([10.88132286, 10.83174439]),
 'radius': 9.000000000000002,
 'dist1': 2.009892716098348,
 'dist2': 11.761811095385543}
```

createSlices()

Method for defining the structure of all the slices in which the soil mass above the slip surface is divided.

Returns List of object instanced from the class `SliceStr` that defines the structure of an individual slice.

Return type (*list*)

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
```

(continues on next page)

(continued from previous page)

```
>>> from pybimstab.slices import MaterialParameters, Slices
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=2, dist2=10, radius=9)
>>> material = MaterialParameters(
>>>     cohesion=15, frictAngle=23, unitWeight=17,
>>>     blocksUnitWeight=21, wtUnitWeight=9.8)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=10,
>>>     watertabCoords=None, bim=None)
>>> slicesList = slices.createSlices()
>>> slicesList[0].__dict__.keys()
dict_keys(['material', 'terrainLS', 'slipSurfLS', 'watertabLS',
           'bim', 'terrainCoords', 'slipSurfCoords',
           'watertabCoords', 'sliceLS', 'coords', 'xMin', 'xMax',
           'yMin', 'yMax', 'area', 'width', 'baseSlope', 'alpha',
           'baseLength', 'l', 'topLength', 'topSlope',
           'topInclinatDeg', 'midHeight', 'midWatTabHeight'])
```

setExtLoads(*extL=[{'load': 0, 'angle': 0}]*)

Method for setting the external loads to the slices.

Parameters **extL** (*list*) – list that stores the information of the external loads in dictionaries. Each dictionary has the value of the external force (at the slope surface) and its inclination in degrees as the following structure `{'load': 0, 'angle': 0}`. There are three possibilities to define the structure:

- Unitary list means that the input is constant for all the slices. `{'load': 0, 'angle': 0}` is the default value.
- If the length of the list is as long as the number of slices, each slice is coupled with the input list.
- If the length of the list is $l > 1$ and lower than the number of slices, then only the l fisrt slices are coupled with the external loads.

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.slices import MaterialParameters, Slices
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=2, dist2=10, radius=9)
>>> material = MaterialParameters(
>>>     cohesion=15, frictAngle=23, unitWeight=17,
>>>     blocksUnitWeight=21, wtUnitWeight=9.8)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=10,
>>>     watertabCoords=None, bim=None)
>>> slices.setExtLoads(extL=[{'load': 200, 'angle': 30}])
```

(continues on next page)

(continued from previous page)

```
>>> slices.slices[0].extL, slices.slices[0].w
(200, 30)
```

plot(*plotFittedCirc=False*)

Method for generating a graphic of the slope stability model when is possible to watch the slices in the soil mass above the slip surface.

Returns object with the matplotlib structure of the plot. You might use it to save the figure for example.

Return type (*matplotlib.figure.Figure*)

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.slices import MaterialParameters, Slices
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=2, dist2=10, radius=9)
>>> material = MaterialParameters(
>>>     cohesion=15, frictAngle=23, unitWeight=17,
>>>     blocksUnitWeight=21, wtUnitWeight=9.8)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=10,
>>>     watertabCoords=None, bim=None)
>>> fig = slices.plot()
```

```
>>> from numpy import array
>>> from pybimstab.slope import NaturalSlope
>>> from pybimstab.watertable import WaterTable
>>> from pybimstab.bim import BlocksInMatrix
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.slipsurface import TortuousSurface
>>> from pybimstab.slices import MaterialParameters, Slices
>>> terrainCoords = array(
>>>     [[-2.49, 0.1, 1.7, 3.89, 5.9, 8.12, 9.87, 13.29, 20.29,
>>>       21.43, 22.28, 23.48, 24.65, 25.17],
>>>      [18.16, 17.88, 17.28, 15.73, 14.31, 13.58, 13, 3.61, 3.61,
>>>       3.32, 2.71, 2.23, 1.21, 0.25]])
>>> slope = NaturalSlope(terrainCoords)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.3,
>>>                      tileSize=0.35, seed=123)
>>> watertabDepths = array([[0, 5, 10, 15],
>>>                         [8, 7, 3, 0]])
>>> watertable = WaterTable(slopeCoords=slope.coords,
>>>                         watertabDepths=watertabDepths,
>>>                         smoothFactor=3)
>>> preferredPath = CircularSurface(
>>>     slopeCoords=slope.coords, dist1=5, dist2=15.78, radius=20)
>>> surface = TortuousSurface(
```

(continues on next page)

(continued from previous page)

```
>>>     bim, dist1=4, dist2=15.78, heuristic='euclidean',
>>>     reverseLeft=False, reverseUp=False, smoothFactor=2,
>>>     preferredPath=preferredPath.coords, prefPathFact=2)
>>> material = MaterialParameters(
>>>     cohesion=15, frictAngle=23, unitWeight=17,
>>>     blocksUnitWeight=21, wtUnitWeight=9.8)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=10,
>>>     watertabCoords=watertable.coords, bim=bim)
>>> fig = slices.plot()
```

2.5 slipsurface

Module for defining the classes related to the slip surface, either a circular or a composite geometry (*e.g.* a tortuous failure surface)

class slipsurface.**CircularSurface**(*slopeCoords*, *dist1*, *dist2*, *radius*, *concave=True*)

Bases: object

Creates an instance of an object that defines the structure of an polyline which represents the slip surface of a landslide which geometry is a circumference-arc.

```
CircularSurface(slopeCoords, dist1, dist2, radius, concave=True)
```

The arc is defined with two points on the terrain surface and the radius. That implies there are two possible solutions; to select which one is wanted, it is necessary to modify the variable `concave`.

It is possible the arc cuts across the terrain surface in some point different to its ends, perhaps because of some swedge in the terrrain or the radius is too long. In that case, the method `defineStructre` changes the attribute `dist2` such that it is replaced by the horizontal distance of the intersection point.

slopeCoords

Coordinates of the vertices of the polygon within which the slope mass is defined. It is obtained with the method `defineboundary` either from the classes `AnthropicSlope` or `NaturalSlope` (module `slope`).

Type (2, n *numpy.ndarray*

dist1

First horizontal distance from the leftmost point of the terrain surface (including the crown) where the arc is intersected with it.

Type *int* or *float*

dist2

Second horizontal distance from the leftmost point of the terrain surface (including the crown) where the arc is intersected with it.

Type *int* or *float*

radius

Length of the circumference-arc radius.

Type *int* or *float*

concave
Logical variable to define if it is wanted that the circumference-arc will be concave (upwards), otherwise, it will be convexe (downwards). Default value is `True`.

Type *bool*

Note: The class `CircularSurface` requires numpy, matplotlib and shapely.

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=2, dist2=10, radius=9)
>>> surface.__dict__.keys()
dict_keys(['slopeCoords', 'dist1', 'dist2', 'radius', 'concave',
           'point1', 'point2', 'center', 'initAngle',
           'endAngle', 'coords'])
```

defineStructre()
Method to define the structure of the circumference-arc which represents the slip surface of a landslide.

If the arc cuts across the terrain surface in some point different to its ends, the attribute `dist2` is modified to the horizontal distance of the intersection point.

The returned angles have values betwen :math: *left[pi, -pi right)*, where the angle equal to zero coincides with the vector :math: *left(1, 0) right)*.

Returns

dictionary with the following outputs.

- **center** (*tuple*): Coordinates of the circumference-arc center
- **endAngle** (*float*): Angle in radians of the vector that points from the center to the first intersection between the terrain surface and the circumference-arc.
- **initAngle** (*float*): Angle in radians of the vector that points from the center to the first intersection between the terrain surface and the circumference-arc.
- **point1** (*tuple*): Coordinates of the first point that intersects the terrain surface
- **point2** (*tuple*): Coordinates of the second point that intersects the terrain surface

Return type (*dict*)

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
```

(continues on next page)

(continued from previous page)

```
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=2, dist2=10, radius=9)
>>> surface.defineStructre()
{'center': (10.881322862689261, 10.831744386868543),
 'endAngle': -1.668878272858519,
 'initAngle': -2.979016942655663,
 'point1': array([2.   , 9.375]),
 'point2': array([10.   ,  1.875])}
```

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=2, dist2=10, radius=1)
>>> surface.defineStructre()
ValueError: separation of points > diameter
```

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=2, dist2=10, radius=6)
>>> surface.defineStructre()
ValueError: Radius too short. Increase at least 1.516
```

plot()

Method for generating a graphic of the circumference-arc and the slope.

Returns object with the matplotlib structure of the plot. You might use it to save the figure for example.

Return type (*matplotlib.figure.Figure*)

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=2, dist2=10, radius=9)
>>> fig = surface.plot()
```

```
>>> from numpy import array
>>> from pybimstab.slope import NaturalSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> terrainCoords = array(
>>>     [[-2.49, 0.1 , 1.7 , 3.89, 5.9 , 8.12, 9.87, 13.29, 20.29,
```

(continues on next page)

(continued from previous page)

```
>>>       21.43, 22.28, 23.48, 24.65, 25.17],
>>>      [18.16, 17.88, 17.28, 15.73, 14.31, 13.58, 13, 3.61, 3.61,
>>>       3.32, 2.71, 2.23, 1.21, 0.25]])
>>> slope = NaturalSlope(terrainCoords)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=7, dist2=20, radius=13)
>>> fig = surface.plot()
```

class slipsurface.**TortuousSurface**(*bim*, *dist1*, *dist2*, *heuristic='manhattan'*, *reverseLeft=False*, *reverseUp=False*, *smoothFactor=0*, *preferredPath=None*, *prefPathFact=None*)

Bases: `object`

Creates an instance of an object that defines the structure of an polyline which represents the slip surface of a landslide which geometry is a tortuous path surrouding the blocks inside the slope mass.

```
TortuousSurface(bim, dist1, dist2, heuristic='Manhattan',
                reverseLeft=False, reverseUp=False, smoothFactor=0,
                preferredPath=None, prefPathFact=None)
```

The surface is defined with two points on the terrain surface and the heuristic function. It is possible to set a forced path to modify the free trajectory of the tortuous path with the aim of move it closer to, for example a circular surface.

bim
: object with the structure of the slope made of the Blocks-In-Matrix material.

  Type *BlocksInMatrix* object

dist1
: First horizontal distance from the leftmost point of the terrain surface (including the crown) where the arc is intersected with it.

  Type *int* or *float*

dist2
: Second horizontal distance from the leftmost point of the terrain surface (including the crown) where the arc is intersected with it.

  Type *int* or *float*

heuristic
: Name of the geometric model to determine the heuristic distance. It must be selected either `Manhattan` or `Euclidean`; their description can be found in the `Astar` class documentation. *Manhattan* is the default value.

  Type *str*

reverseLeft
: Logical variable to allow or not reverses movements to the left. Default value is `False`.

  Type *bool*

reverseUp
: Logical variable to allow or not reverses movements to upward. Default value is `False`.

  Type *bool*

smoothFactor
Value to indicate the B-spline interpolation order of the smooter function. If is equal to zero, which is the default value, the surface will not be smoothed.

Type *int*

preferredPath
(2, n) array with the coordinates of a path where the tortuous surface is going to be forced; `None` is the default value.

Type *numpy.ndarray* or *None*

prefPathFact
Multiplier of the shortest distance between the current point and the polyline; `None` is the default value.

Type *int* or *float* or *None*

Note: The class `TortuousSurface` requires numpy, matplotlib and shapely.

Examples

```
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> from pybimstab.slipsurface import TortuousSurface
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
>>>                        crownDist=10, toeDist=10)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.25,
>>>                      tileSize=0.25, seed=123)
>>> surface = TortuousSurface(
>>>     bim, dist1=0, dist2=17, heuristic='manhattan',
>>>     reverseLeft=False, reverseUp=False, smoothFactor=0,
>>>     preferredPath=None, prefPathFact=None)
>>> surface.__dict__.keys()
dict_keys(['bim', 'dist1', 'dist2', 'heuristic', 'reverseLeft',
           'reverseUp', 'smoothFactor', 'preferredPath',
           'prefPathFact', 'terrainSurfLS', 'point1', 'end1', 'point2',
           'end2', 'startIdx', 'goalIdx', 'coords'])
```

getIndexes(*coord*)
Method for obtaining the array indexes of the BIM structure for a coordinate given in the real scale of the slope stability problem.

The transformation is performed by rounding the division between the coordinate and the tile size with the `int_` function of *numpy*. That means that always rounds to the left and bottom sides of a tile.

coord
Coordinates of some point in the slope mass or surface, which is wanted to get them indexes into the BIM grid-grapth structure.

Type *tuple*

Returns Indexes of a coordinate from the real-scale problem projected to the array that represents the BIM structure of the slope; the first value of the tuple is the row and the second one is the column of the grid-array respectively.

Return type (*tuple*)

Examples

```
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> from pybimstab.slipsurface import TortuousSurface
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
>>>                        crownDist=10, toeDist=10)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.25,
>>>                      tileSize=0.25, seed=123)
>>> surface = TortuousSurface(bim, dist1=0, dist2=17)
>>> surface.getIndexes(surface.point1)
(66, 00)
>>> surface.getIndexes(surface.point2)
(24, 68)
```

getCoord(*indexes*)

Method for obtaining the real scale problem coordinates of of some cell in the BIM grid-graph structure of the slope.

The transformation is performed by getting the center of the tile which contains the coordinates of the point.

indexes

Indexes of some cell in the BIM grid-graph structure of the slope; the first tuple-value is the ordinate and the second one is the abscisse.

Type *tuple*

Returns Indexes of a coordinate from the real-scale problem projected to the array that represents the BIM structure of the slope; the first tuple value is the abscisse and the second one is the ordinate.

Return type (*tuple*)

Examples

```
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> from pybimstab.slipsurface import TortuousSurface
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
>>>                        crownDist=10, toeDist=10)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.25,
>>>                      tileSize=0.25, seed=123)
>>> surface = TortuousSurface(bim, dist1=0, dist2=17)
>>> surface.getCoord((66, 0))
(0.125, 16.446428571428573)
>>> surface.getCoord((24, 68))
(17.125, 5.946428571428573)
```

getIndexesAtEnds()

Method for obtaining the array indexes of the BIM grid-grapth structure for the ends of the slip surface.

Examples

```
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> from pybimstab.slipsurface import TortuousSurface
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
>>>                        crownDist=10, toeDist=10)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.25,
>>>                      tileSize=0.25, seed=123)
>>> surface = TortuousSurface(bim, dist1=0, dist2=17)
>>> surface.getIndexesAtEnds()
((66, 0), (23, 68))
```

defineStructre()

Method to define the structure of the tortuous path which represents the slip surface of a landslide that occurs in a slope made of BIM.

The surface is generated through the A* algorithm defined in the `Astar` module.

Examples

```
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> from pybimstab.slipsurface import TortuousSurface
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
>>>                        crownDist=10, toeDist=10)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.25,
>>>                      tileSize=0.25, seed=123)
>>> surface = TortuousSurface(bim, dist1=0, dist2=17)
>>> surface.defineStructre()
array([[ 0.        ,  0.125     ,  0.375     ,  0.625     , ... ],
       [16.57142857, 16.44642857, 16.19642857, 15.94642857, ... ]])
```

plot()

Method for generating a graphic of the tortuous slip surface and the slope.

Returns object with the matplotlib structure of the plot. You might use it to save the figure for example.

Return type (*matplotlib.figure.Figure*)

Examples

```
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> from pybimstab.slipsurface import TortuousSurface
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
>>>                        crownDist=10, toeDist=10)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.25,
>>>                      tileSize=0.25, seed=123)
```

```
>>> # Not allowing to turn left and up
>>> surface = TortuousSurface(
>>>     bim, dist1=0, dist2=17, heuristic='manhattan',
>>>     reverseLeft=False, reverseUp=False, smoothFactor=0,
>>>     preferredPath=None, prefPathFact=None)
>>> fig = surface.plot()
```

```
>>> # Allowing to turn left and up (manhattan heusitic function)
>>> surface = TortuousSurface(
>>>     bim, dist1=0, dist2=17, heuristic='manhattan',
>>>     reverseLeft=True, reverseUp=True, smoothFactor=0,
>>>     preferredPath=None, prefPathFact=None)
>>> fig = surface.plot()
```

```
>>> from numpy import array
>>> from pybimstab.slope import NaturalSlope
>>> from pybimstab.bim import BlocksInMatrix
>>> from pybimstab.slipsurface import CircularSurface, TortuousSurface
>>> terrainCoords = array(
>>>     [[-2.49, 0.1, 1.7, 3.89, 5.9, 8.12, 9.87, 13.29, 20.29,
>>>       21.43, 22.28, 23.48, 24.65, 25.17],
>>>      [18.16, 17.88, 17.28, 15.73, 14.31, 13.58, 13, 3.61, 3.61,
>>>       3.32, 2.71, 2.23, 1.21, 0.25]])
>>> slope = NaturalSlope(terrainCoords)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.3,
>>>                      tileSize=0.35, seed=123)
>>> preferredPath = CircularSurface(
>>>     slopeCoords=slope.coords, dist1=5, dist2=15.78, radius=20)
```

```
>>> # With a preferred path and smoothing the surface
>>> surface = TortuousSurface(
>>>     bim, dist1=4, dist2=15.78, heuristic='euclidean',
>>>     reverseLeft=False, reverseUp=False, smoothFactor=2,
>>>     preferredPath=preferredPath.coords, prefPathFact=2)
>>> fig = surface.plot()
```

```
>>> # Without a preferred path and smoothing the surface
>>> surface = TortuousSurface(
>>>     bim, dist1=5, dist2=15.78, heuristic='euclidean',
>>>     reverseLeft=False, reverseUp=False, smoothFactor=2,
>>>     preferredPath=None)
>>> fig = surface.plot()
```

2.6 slope

Module for defining the class related to the slope geometry.

class slope.**AnthropicSlope**(*slopeHeight*, *slopeDip*, *crownDist*, *toeDist*, *depth=None*)

Bases: object

Creates an instance of an object that defines the geometrical frame of the slope to perform the analysis given the geometric properties of the slope.

```
AnthropicSlope(slopeHeight, slopeDip, crownDist, toeDist, depth=None)
```

The geometry of the slope is as follow:

- It is a right slope, i.e. its face points to the right side.
- Crown and toe planes are horizontal.
- The face of the slope is continuous, ie, it has not berms.

slopeHeight
Height of the slope, ie, vertical length betwen crown and toe planes.

Type *int* or *float*

slopeDip
Both horizontal and vertical components of the slope inclination given in that order.

Type (2,) *tuple*, *list* or *numpy.ndarray*

crownDist
Length of the horizontal plane in the crown of the slope.

Type *int* or *float*

toeDist
Length of the horizontal plane in the toe of the slope.

Type *int* or *float*

depth
Length of the segment beneath the slope toe. `None` is the default value; if is None, then the maximum depth is calculated.

Type *int* or *float* or *None*

Note: The class `AnthropicSlope` requires numpy and matplotlib.

Examples

```
>>> from pybimstab.slope import AnthropicSlope
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
>>>                        crownDist=10, toeDist=10)
>>> slope.__dict__.keys()
{'coords': array(
        [[ 0.        ,  0.        , 10.        , 18.        ,
          28.        , 28.        ,  0.        ],
         [ 0.        , 16.57142857, 16.57142857,  4.57142857,
           4.57142857,  0.        ,  0.        ]]),
 'crownDist': 10,
 'depth': 4.571428571428573,
 'slopeDip': array([1. , 1.5]),
 'slopeHeight': 12,
 'toeDist': 10}
```

maxDepth()
Method to obtain the maximum depth of a slope where a circular slope failure analysis can be performed.

The maximum depth is such that the biggest circle satisfished the following conditions:

- It is tangent to the bottom.

- crosses both the extreme points at the crown and toe.
- It is orthogonal to the crown plane.

Returns Maximum depth of the slope measured vertically from the toe plane.

Return type (*int* or *float*)

Examples

```
>>> from pybimstab.slope import AnthropicSlope
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
                           crownDist=10, toeDist=10)
>>> slope.maxDepth()
4.571428571428573
```

defineBoundary()

Method to obtain the coordinates of the boundary vertices of the slope and plot it if it is wanted.

The origin of the coordinates is in the corner of the bottom with the back of the slope. The coordinates define a close polygon, ie, the first pair of coordinates is the same than the last one.

Returns Coordinates of the boundary vertices of the slope.

Return type (*numpy.ndarray*)

Examples

```
>>> from pybimstab.slope import AnthropicSlope
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
>>>                        crownDist=10, toeDist=10)
>>> slope.defineBoundary()
array([[ 0.   ,  0.   , 10.   , 18.   , 28.   , 28.   ,  0.   ],
       [ 0.   , 16.571, 16.571,  4.571,  4.571,  0.   ,  0.   ]])
```

plot()

Method for generating a graphic of the slope boundary.

Returns object with the matplotlib structure of the plot. You might use it to save the figure for example.

Return type (*matplotlib.figure.Figure*)

Examples

```
>>> from pybimstab.slope import AnthropicSlope
>>> slope = AnthropicSlope(slopeHeight=12, slopeDip=[1, 1.5],
>>>                        crownDist=10, toeDist=10)
>>> fig = slope.plot()
```

class slope.**NaturalSlope**(*terrainCoords*, *depth=None*)

Bases: object

Creates an instance of an object that defines the geometrical frame of the slope to perform the analysis given the terrain coordinates.

```
NaturalSlope(terrainCoords, depth=None)
```

The geometry of the slope is as follow:

- It is a right slope, i.e. its face points to the right side.
- The slope is defined with its surface's coordinates.
- The surface is defined as a polyline such that each segment's slope are always zero or negative.
- The coordinates' order is such that the highest (and leftmost) point is the first one, and the lowest (and rightmost) is the last one.

terrainCoords
(2, n) array with the coordinates of the slope surface. It must not be a closed polyline, just the terrain segment.

Type *numpy.ndarray*

depth
Length of the segment beneath the slope toe. `None` is the default value; if is None, then the maximum depth is calculated.

Type *int* or *float* or *None*

Note: The class `NaturalSlope` requires numpy and matplotlib.

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import NaturalSlope
>>> coords = array(
>>>     [[ 0.   , 28.   , 28.   , 18.   , 10.   ,  0.   ],
>>>      [ 0.   ,  0.   ,  4.571,  4.571, 16.571, 16.571]])
>>> slope = NaturalSlope(coords)
>>> slope.__dict__.keys()
dict_keys(['coords', 'slopeHeight', 'maxDepth', 'coords'])
```

maxDepth()
Method to obtain the maximum depth of a slope where a circular slope failure analysis can be performed.

The maximum depth is such that the biggest circle satisfished the following conditions:

- It is tangent to the bottom.
- crosses both the extreme points at the crown and toe.
- It is orthogonal to the crown plane.

Returns Maximum depth of the slope measured vertically from the rightmost point of the surface coordinates.

Return type (*int* or *float*)

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import NaturalSlope
>>> terrainCoords = array(
>>>     [[0, 10, 18, 28], [16.571, 16.571,  4.571,  4.571]])
>>> slope = NaturalSlope(terrainCoords)
>>> slope.maxDepth()
4.571428571428571
```

defineBoundary()

Method to obtain the coordinates of the boundary vertices of the slope and plot it if it is wanted.

The origin of the coordinates is in the corner of the bottom with the back of the slope. The coordinates define a closed polygon, ie, the first pair of coordinates is the same than the last one. That closed polygon is sorted clockwise.

Returns Coordinates of the boundary vertices of the slope.

Return type (*numpy.ndarray*)

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import NaturalSlope
>>> terrainCoords = array(
>>>     [[0, 10, 18, 28], [16.571, 16.571,  4.571,  4.571]])
>>> slope = NaturalSlope(terrainCoords)
>>> slope.defineBoundary()
array([[ 0.  ,  0.  , 10.  , 18.  , 28.  , 28.  ,  0.  ],
       [ 0.  , 16.57, 16.57,  4.57,  4.57,  0.  ,  0.  ]])
```

```
>>> from numpy import array
>>> terrainCoords = array(
>>>     [[-2.49, 0.1 , 1.7 , 3.89, 5.9 , 8.12, 9.87, 13.29, 20.29,
>>>       21.43, 22.28, 23.48, 24.65, 25.17],
>>>      [18.16, 17.88, 17.28, 15.73, 14.31, 13.58, 13, 3.61, 3.61,
>>>       3.32, 2.71, 2.23, 1.21, 0.25]])
>>> slope = NaturalSlope(terrainCoords)
>>> slope.defineBoundary()
array([[0, 0,  2.59, 4.19, 6.38,  8.39, 10.61, 12.36, 15.78, 22.78,
        23.92, 24.77, 25.97, 27.14, 27.66, 27.66, 0],
       [0, 19.63, 19.35, 18.75, 17.2 , 15.78, 15.05, 14.47,  5.08,
        5.08, 4.79, 4.18, 3.7 , 2.68,  1.72, 0, 0]])
```

plot()

Method for generating a graphic of the slope boundary.

Returns object with the matplotlib structure of the plot. You might use it to save the figure for example.

Return type (*matplotlib.figure.Figure*)

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import NaturalSlope
>>> terrainCoords = array(
>>>     [[0, 10, 18, 28], [16.571, 16.571,  4.571,  4.571]])
>>> slope = NaturalSlope(terrainCoords)
>>> fig = slope.plot()
```

```
>>> from numpy import array
>>> from pybimstab.slope import NaturalSlope
>>> terrainCoords = array(
>>>     [[-2.49, 0.1 , 1.7 , 3.89, 5.9 , 8.12, 9.87, 13.29, 20.29,
>>>       21.43, 22.28, 23.48, 24.65, 25.17],
>>>      [18.16, 17.88, 17.28, 15.73, 14.31, 13.58, 13, 3.61, 3.61,
>>>       3.32, 2.71, 2.23, 1.21, 0.25]])
>>> slope = NaturalSlope(terrainCoords)
>>> fig = slope.plot()
```

2.7 slopestabl

Module for evaluating the factor of safety against sliding by using the limit equilibrium method through the General Limit Equilibrium (GLE) method presented by Fredlund & Krahn (1977).

class slopestabl.**SlopeStabl**(*slices*, *seedFS=1*, *Kh=0*, *maxIter=50*, *tol=0.001*, *interSlcFunc='halfsine'*, *minLambda=-0.6*, *maxLambda=0.6*, *nLambda=10*)

Bases: object

Creates an instance of an object that allows to evaluate the factor of safety against sliding of a slope.

```
SlopeStabl(slices, seedFS=1, Kh=0, maxIter=50, tol=1e-3,
           interSlcFunc='halfsine', minLambda=-0.6, maxLambda=0.6,
           nLambda=10)
```

Attributes:

slices (*Slices* object): object that contains the data structure of the slices in which the sliding mass has been divided.

seedFS (*float* or *int*): Initial value of factor of safety for starting the iterarive algorithm. 1 is the default value.

lambda_ (*float* or *int*): Factor that multiplies the interlice function to determine the interslices horizontal forces. 0 is the default value.

Kh (*float*): horizontal seismic coefficient for the pseudostatic analysis. Its positive value represents the force is directed out of the slope (i.e. in the direction of failure). 0 is the default value.

maxIter (*int*): Maximum number of iterations for stoping the algorithm in case the tolerance is not reached. 50 is the default value.

tol (*float*): Required tolerace to stop the iterations. Is the diference between the 2 last values gotten of factor of safety and lambda, it means, two tolerances have to be reached. `1e-3` is the default value.

interSlcFunc (*str* or 'float'): Interslice function that relates the normal interslice forces and the parameter lambda to obtain the shear interslice forces. `halfsine` is the default value and corresponds to Morgenstern and Price method, but a constant number may be input, for example `interSlcFunc=1`, corresponds to Spencer method.

maxLambda (*float*): Maximum value the lambda parameter can get. `0.6` is the default value.

nLambda (*float*): Number of value the lambda parameter can get from zero to `maxLambda`. `6` is the default value.

slices, seedFS=1, Kh=0, maxIter=50, tol=1e-3,

interSlcFunc='halfsine', maxLambda=0.6, nLambda=6

Note: The class `Slices` requires numpy, scipy, matplotlib and shapely.

Examples:

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.watertable import WaterTable
>>> from pybimstab.slices import MaterialParameters, Slices
>>> from pybimstab.slopestabl import SlopeStabl
>>> slope = AnthropicSlope(slopeHeight=40, slopeDip=[2, 1],
>>>                        crownDist=60, toeDist=30, depth=20)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=45.838, dist2=158.726, radius=80)
>>> material = MaterialParameters(
>>>     cohesion=600, frictAngle=20, unitWeight=120, wtUnitWeight=62.4)
>>> watertable = WaterTable(slopeCoords=slope.coords,
                            watertabDepths=array([[0, 140], [20, 0]]))
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=50,
>>>     watertabCoords=watertable.coords, bim=None)
>>> stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0)
>>> stabAnalysis.__dict__.keys()
dict_keys(['slices', 'Kh', 'seedFS', 'maxIter', 'tol', 'interSlcFunc',
           'minLambda', 'maxLambda', 'nLambda', 'fsBishop',
           'fsJanbu', 'fsMoment', 'fsForces', 'lambda_',
           'adjustment', 'FS'])
```

`intersliceForceFunct` (*v=1*, *u=1*)

Method for calculating the interslice function which is a component of the interslice forces; this is done by using the Equation [11] of Zhu et al (2015), with v = u = 1 for a simetric and non-narrowed halfsine function.

When the object is instanced with the clases with a constant interslice function, then, all the values are equal to that constant value.

Parameters

- **`v`** (*int* or *float*) – shape parameter. Controls the symmetry. `1` is the defaut value.
- **`u`** (*int* or *float*) – shape parameter. Controls the kurtosis. `1` is the defaut value.

Returns Values of the all insterslice force function values.

Return type (*list*)

Examples

```
>>> import matplotlib.pyplot as plt
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.slices import MaterialParameters, Slices
>>> from pybimstab.slopestabl import SlopeStabl
>>> slope = AnthropicSlope(slopeHeight=40, slopeDip=[2, 1],
>>>                        crownDist=60, toeDist=30, depth=20)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=45.838, dist2=158.726,
>>>                           radius=80)
>>> material = MaterialParameters(cohesion=600, frictAngle=20,
>>>                               unitWeight=120,
>>>                               wtUnitWeight=62.4)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=50)
>>> stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0)
>>> interslcForceFunc = stabAnalysis.intersliceForceFunct(u=1)
>>> plt.plot(interslcForceFunc, 'k')
```

calculateArms()

Method for calculating the arms required for getting the momments of each slice with respect to a rotation point.

This function does not return any output, just modifies the structure of each slice by setting new attributes.

calculateBasicForces()

Method for calculating the forces that do not vary in each iteration or lambda value.

This function does not return any output, just modifies the structure of each slice by setting new attributes.

calculateNormalForce(*seedFS*, *fellenius=False*)

Method for calculating the normal force to the base; this is done by using the Equation of section 14.6 of GeoSlope (2015)

Since the normal forces are updated with each iteration, is necessary to input a factor of safety as a seed.

Parameters **seedFS** (*int* or *float*) – Seed factor of safety.

Returns Values of all the normal forces at the slice's bases

Return type (*list*)

Examples

```
>>> from numpy import array
>>> import matplotlib.pyplot as plt
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.watertable import WaterTable
```

(continues on next page)

(continued from previous page)

```
>>> from pybimstab.slices import MaterialParameters, Slices
>>> from pybimstab.slopestabl import SlopeStabl
>>> slope = AnthropicSlope(slopeHeight=40, slopeDip=[2, 1],
>>>                        crownDist=60, toeDist=30, depth=20)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=45.838, dist2=158.726,
>>>                           radius=80)
>>> material = MaterialParameters(cohesion=600, frictAngle=20,
>>>                               unitWeight=120,
>>>                               wtUnitWeight=62.4)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=5)
>>> stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0)
>>> stabAnalysis.calculateNormalForce(stabAnalysis.FS['fs'])
[45009.409630951726, 68299.77910530512, 70721.13554871723,
 57346.7578530581, 22706.444365285253]
```

getFm(*seedFS*, *lambda_=0*, *fellenius=False*)

Method for getting the factor of safety with respect to the moments equilimrium; this is done by using the Equation [22] of Fredlund & Krahn (1977).

Since the factor of safety is updated with each iteration, is necessary to input a factor of safety as a seed and the current value of lambda to relate the interslice normal force and the interslice force function with respect to the interslice shear force (Eq. [16] of Fredlund & Krahn (1977)).

Parameters

- **seedFS** (*int* or *float*) – Seed factor of safety.
- **lambda** (*int* or *float*) – Seed value of lambda. `0` is the default value.

Returns Dictionary with the value of the factor of safety and a tuple with the boolean that indicated if the toleranfe was reached and the number of the iteration.

Return type (*dict*)

Examples

```
>>> # Example Case 1 - Fig. 9 (Fredlund & Krahn, 1977)
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.slices import MaterialParameters, Slices
>>> from pybimstab.slopestabl import SlopeStabl
>>> slope = AnthropicSlope(slopeHeight=40, slopeDip=[2, 1],
>>>                        crownDist=60, toeDist=30, depth=20)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=45.838, dist2=158.726,
>>>                           radius=80)
>>> material = MaterialParameters(cohesion=600, frictAngle=20,
>>>                               unitWeight=120,
>>>                               wtUnitWeight=62.4)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=50,
>>>     watertabCoords=None, bim=None)
>>> stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0, minLambda=0,
```

(continues on next page)

(continued from previous page)

```
>>>                             interSlcFunc=1, nLambda=10)
>>> stabAnalysis.getFm(stabAnalysis.FS['fs'],
>>>                    stabAnalysis.FS['lambda'])
(2.0750390044795854, True)
```

getFf (*seedFS*, *lambda_=0*)

Method for getting the factor of safety with respect to the forces equilimrium; this is done by using the Equation [23] of Fredlund & Krahn (1977).

Since the factor of safety is updated with each iteration, is necessary to input a factor of safety as a seed and the current value of lambda to relate the interslice normal force and the interslice force function with respect to the interslice shear force (Eq. [16] of Fredlund & Krahn (1977)).

Parameters

- **seedFS** (*int* or *float*) – Seed factor of safety.
- **lambda** (*int* or *float*) – Seed value of lambda. 0 is the default value.

Returns Dictionary with the value of the factor of safety and a tuple with the boolean that indicated if the toleranfe was reached and the number of the iteration.

Return type (*dict*)

Examples

```
>>> # Example Case 1 - Fig. 9 (Fredlund & Krahn, 1977)
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.slices import MaterialParameters, Slices
>>> from pybimstab.slopestabl import SlopeStabl
>>> slope = AnthropicSlope(slopeHeight=40, slopeDip=[2, 1],
>>>                        crownDist=60, toeDist=30, depth=20)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=45.838, dist2=158.726,
>>>                           radius=80)
>>> material = MaterialParameters(cohesion=600, frictAngle=20,
>>>                               unitWeight=120,
>>>                               wtUnitWeight=62.4)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=50,
>>>     watertabCoords=None, bim=None)
>>> stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0, minLambda=0,
>>>                           interSlcFunc=1, nLambda=10)
>>> stabAnalysis.getFf(stabAnalysis.FS['fs'],
>>>                    stabAnalysis.FS['lambda'])
(2.0741545445738296, True)
```

intersliceForces (*seedFS*, *lambda_*)

Method for getting the shear and normal interslice forces; this is done by using the Equation of section 14.8 of GeoSlope (2015) for the rigth normal force and the Equation [18] of Fredlund & Krahn (1977) for the shear force.

Since the interslice forces are updated with each iteration, is necessary to input a factor of safety as a seed and the current value of lambda to relate the interslice normal force and the interslice force function with respect to the interslice shear force (Eq. [20] of Fredlund & Krahn (1977)).

Parameters

- **seedFS** (*int* or *float*) – Seed factor of safety.
- **lambda** (*int* or *float*) – Seed value of lambda. `0` is the default value.

Returns tuple with the interslice forces. the first element contains the normal interslice forces and the second contains the shear interslice forces.

Return type (*tuple*)

Examples

```
>>> from numpy import array
>>> import matplotlib.pyplot as plt
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.watertable import WaterTable
>>> from pybimstab.slices import MaterialParameters, Slices
>>> from pybimstab.slopestabl import SlopeStabl
>>> slope = AnthropicSlope(slopeHeight=40, slopeDip=[2, 1],
>>>                        crownDist=60, toeDist=30, depth=20)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=45.838, dist2=158.726,
>>>                           radius=80)
>>> material = MaterialParameters(cohesion=600, frictAngle=20,
>>>                               unitWeight=120,
>>>                               wtUnitWeight=62.4)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=5)
>>> stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0)
>>> stabAnalysis.intersliceForces(stabAnalysis.FS['fs'],
>>>                               stabAnalysis.FS['lambda'])
([0, -24561.260979675248, -42085.32887504204, -38993.844201424305,
  -18464.723052348225, -61.4153504520018],
 [0, -5511.202498703704, -15279.673506543182, -14157.266298947989,
  -4143.22489013017, -2.8712090198929304e-15])
```

iterateGLE()

Method for getting the factor of safety against sliding through the algorithm of the General Limit Equilibrium (GLE) proposed by Fredlund & Krahn (1977)).

Returns factor of safety against sliding is the solution exists.

Return type (*tuple* or *None*)

Examples

```
>>> from numpy import array
>>> import matplotlib.pyplot as plt
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.watertable import WaterTable
>>> from pybimstab.slices import MaterialParameters, Slices
>>> from pybimstab.slopestabl import SlopeStabl
>>> slope = AnthropicSlope(slopeHeight=40, slopeDip=[2, 1],
```

(continues on next page)

(continued from previous page)

```
>>>                          crownDist=60, toeDist=30, depth=20)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=45.838, dist2=158.726,
>>>                           radius=80)
>>> material = MaterialParameters(cohesion=600, frictAngle=20,
>>>                               unitWeight=120,
>>>                               wtUnitWeight=62.4)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=5)
```

```
>>> stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0)
>>> stabAnalysis.iterateGLE()
{'fs': 2.0258090954552275, 'lambda': 0.38174822248691215}
```

```
>>> stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0, nLambda=0)
>>> stabAnalysis.iterateGLE()
{'fsBishop': 2.0267026043637175, 'fsJanbu': 1.770864711650081}
```

plot()

Method for generating a graphic of the slope stability analysis, including the plot of the convergences

Returns object with the matplotlib structure of the plot. You might use it to save the figure for example.

Return type (*matplotlib.figure.Figure*)

Examples

```
>>> # Example Case 1 - Fig. 9 (Fredlund & Krahn, 1977)
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.slices import MaterialParameters, Slices
>>> from pybimstab.slopestabl import SlopeStabl
>>> slope = AnthropicSlope(slopeHeight=40, slopeDip=[2, 1],
>>>                        crownDist=60, toeDist=30, depth=20)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=45.838, dist2=158.726,
>>>                           radius=80)
>>> material = MaterialParameters(cohesion=600, frictAngle=20,
>>>                               unitWeight=120,
>>>                               wtUnitWeight=62.4)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=50,
>>>     watertabCoords=None, bim=None)
>>> stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0, minLambda=0,
>>>                           interSlcFunc=1, nLambda=10)
>>> fig = stabAnalysis.plot()
```

```
>>> # Example Case 5 (Fredlund & Krahn, 1977)
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
```

(continues on next page)

(continued from previous page)

```
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.watertable import WaterTable
>>> from pybimstab.slices import MaterialParameters, Slices
>>> from pybimstab.slopestabl import SlopeStabl
>>> slope = AnthropicSlope(slopeHeight=40, slopeDip=[2, 1],
>>>                        crownDist=60, toeDist=30, depth=20)
>>> surface = CircularSurface(slopeCoords=slope.coords,
>>>                           dist1=45.838, dist2=158.726,
>>>                           radius=80)
>>> material = MaterialParameters(cohesion=600, frictAngle=20,
>>>                               unitWeight=120,
>>>                               wtUnitWeight=62.4)
>>> watertable = WaterTable(slopeCoords=slope.coords,
>>>                         watertabDepths=array([[0, 140],
>>>                                               [20, 0]]))
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=50,
>>>     watertabCoords=watertable.coords, bim=None)
>>> stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0, minLambda=0)
>>> fig = stabAnalysis.plot()
```

```
>>> from numpy import array
>>> from pybimstab.slope import NaturalSlope
>>> from pybimstab.watertable import WaterTable
>>> from pybimstab.bim import BlocksInMatrix
>>> from pybimstab.slipsurface import CircularSurface
>>> from pybimstab.slipsurface import TortuousSurface
>>> from pybimstab.slices import MaterialParameters, Slices
>>> from pybimstab.slopestabl import SlopeStabl
>>> terrainCoords = array(
>>>     [[-2.49, 0.1, 1.7, 3.89, 5.9, 8.12, 9.87, 13.29, 20.29,
>>>       21.43, 22.28, 23.48, 24.65, 25.17],
>>>      [18.16, 17.88, 17.28, 15.73, 14.31, 13.58, 13, 3.61, 3.61,
>>>       3.32, 2.71, 2.23, 1.21, 0.25]])
>>> slope = NaturalSlope(terrainCoords)
>>> bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.2,
>>>                      tileSize=0.35, seed=3210)
>>> watertabDepths = array([[0, 5, 10, 15],
>>>                         [8, 7, 3, 0]])
>>> watertable = WaterTable(slopeCoords=slope.coords,
>>>                         watertabDepths=watertabDepths,
>>>                         smoothFactor=3)
>>> preferredPath = CircularSurface(
>>>     slopeCoords=slope.coords, dist1=5, dist2=15.78, radius=20)
>>> surface = TortuousSurface(
>>>     bim, dist1=4, dist2=15.5, heuristic='euclidean',
>>>     reverseLeft=False, reverseUp=False, smoothFactor=2,
>>>     preferredPath=preferredPath.coords, prefPathFact=2)
>>> material = MaterialParameters(
>>>     cohesion=15, frictAngle=23, unitWeight=17,
>>>     blocksUnitWeight=21, wtUnitWeight=9.8)
>>> slices = Slices(
>>>     material=material, slipSurfCoords=surface.coords,
>>>     slopeCoords=slope.coords, numSlices=20,
```

(continues on next page)

(continued from previous page)

```
>>>     watertabCoords=watertable.coords, bim=bim)
>>> stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0, nLambda=13,
>>>                           minLambda=0)
>>> fig = stabAnalysis.plot()
```

2.8 smoothcurve

Module for defining the class related to the curve softener.

class smoothcurve.**SmoothCurve**(*x*, *y*, *k=3*, *n=300*)

Bases: object

Creates an instance of an object that defines a curve smoother than the input through the k-order B-Spline method for interpolation.

```
SmoothCurve(x, y, k=3, n=300)
```

x

abscisses of the curve to smooth.

Type *tuple*, *list* or *numpy.ndarray*

y

ordinates of the curve to smooth. It must have the same length of `x`.

Type *tuple*, *list* or *numpy.ndarray*

k

interpolation order.

Type *int*

n

number of points of the returned smooth curve

Type *int*

Note: The class `SmoothCurve` requires numpy, matplotlib. and scipy.

Examples

```
>>> from pybimstab.smoothcurve import SmoothCurve
>>> x = [9, 9, 8, 7, 6, 5, 4, 3, 2, 1, 1, 1, 0, 0, 0, 0]
>>> y = [9, 8, 7, 7, 8, 8, 8, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> curve = SmoothCurve(x=x, y=y)
>>> curve.__dict__.keys()
dict_keys(['x', 'y', 'k', 'n', 'smoothing'])
```

smooth()

Method to generate a smooth curve from the points input through the k-order B-Spline method.

Returns $(2 \times n)$ array where n is the number of nodes where the path has crossed; the first row of the array contains the abscisses and the second one contains the ordinates of the nodes into the grid-graph.

Return type (*numpy.ndarray*)

Examples

```
>>> from pybimstab.smoothcurve import SmoothCurve
>>> x = [9, 9, 8, 7, 6, 5, 4, 3, 2, 1, 1, 1, 0, 0, 0, 0]
>>> y = [9, 8, 7, 7, 8, 8, 8, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> curve = SmoothCurve(x=x, y=y, n=10)
>>> curve.smooth()
array([[9.        , 7.56984454, 6.11111111, 4.66666667, 3.2222222,
        1.77960677, 1.00617284, 0.77206219, 0.01463192, 0.       ],
       [9.        , 7.05749886, 7.77206219, 8.        , 7.9215821,
        6.77777778, 5.33333333, 3.88888889, 2.43015546, 0.       ]])
```

plot()

Method for generating a graphic of the `smooth` method output. It allows visually to compare the no smoothed line and its smooothed version.

Returns object with the matplotlib structure of the plot. You might use it to save the figure for example.

Return type (*matplotlib.figure.Figure*)

Examples

```
>>> from pybimstab.smoothcurve import SmoothCurve
>>> x = [9, 9, 8, 7, 6, 5, 4, 3, 2, 1, 1, 1, 0, 0, 0, 0]
>>> y = [9, 8, 7, 7, 8, 8, 8, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> curve = SmoothCurve(x, y, 1)
>>> fig = curve.plot()
```

```
>>> import numpy as np
>>> from pybimstab.smoothcurve import SmoothCurve
>>> x = np.linspace(0, 2*np.pi, 50)
>>> y = np.sin(x) + np.random.random(50) * 0.5
>>> for k in [0, 2, 15]:
>>>     curve = SmoothCurve(x, y, k)
>>>     fig = curve.plot()
```

2.9 tools

Module for defining the repetitive functions to manipulate the shapely objects that are used in the classes for performing the slope stabity analysis.

tools.**getIntersect**(*x*, *y1*, *y2*)

Intersects two lines that have the same abscise points

Parameters

- **x** (*list* or *tuple*) – abscises of both lines
- **y1** (*list* or *tuple*) – ordinates of first line
- **y2** (*list* or *tuple*) – ordinates of second line

Returns Coordinates of the interesection point

Return type (*tuple*)

Examples

```
>>> from pybimstab.tools import getIntersect
>>> getIntersect(x=[0, 1], y1=[0, 0], y2=[1, -1])
(0.5, 0.0)
```

tools.**cutLine**(*line*, *distance*)

Cuts a line in two parts at a distance measured from its starting point.

Parameters

- **line** (*LineString*) – Any line or polyline from the `shapely` module.
- **distance** (*int* or *float*) – The absolute distance measured from the starting point of the line.

Returns Two lines that merged at the common point conform the original input line.

Return type (*tuple*)

Examples

```
>>> from shapely.geometry import LineString
>>> from pybimstab.tools import cutLine
>>> line = LineString([(0, 0), (5, 0)])
>>> line1, line2 = cutLine(line, 2.5)
>>> list(line1.coords)
[(0.0, 0.0), (2.5, 0.0)]
>>> list(line2.coords)
[(2.5, 0.0), (5.0, 0.0)]
```

tools.**upperLowerSegm**(*line1*, *line2*)

Splits two polylines ath their intersection points and join the upper segments in a new polyline and the lower segments in other

Parameters

- **line1** (*LineString*) – Any line or polyline from the `shapely` module.
- **line2** (*LineString*) – Any line or polyline from the `shapely` module.

Returns Two lines where each one is given in a (n, 2) array with the coordinates. The first one is the outcome of joinning the upper segments and the second is the outcome of joinning the lower segments.

Return type (*tuple*)

Examples

```
>>> from shapely.geometry import LineString
>>> from pybimstab.tools import upperLowerSegm
>>> line1 = LineString([(0, 0), (5, 0)])
>>> line2 = LineString([(0, 5), (5, -5)])
>>> upperLine, lowerLine = upperLowerSegm(line1, line2)
>>> upperLine, lowerLine
(array([[ 0. ,  0. ],
        [ 2.5,  0. ],
        [ 2.5,  0. ],
        [ 5. , -5. ]]),
 array([[0. , 5. ],
        [2.5, 0. ],
        [2.5, 0. ],
        [5. , 0. ]]))
```

tools.**getPointAtX**(*line*, *x*)

Interesects a vertical line at the abscice `x` with the input line and retuns the intersection point.

The input line must not have more than one intersection with the vertical line.

If the abscise is out of the horizontal range of the line, then the nearest end is returned.

Parameters

- **line** (*LineString*) – Any line or polyline from the `shapely` module.
- **x** (*int* or *float*) – The abscise where the vertical line is wanted.

Returns Two lines where each one is given in a (n, 2) array with the coordinates. The first one is the outcome of joinning the upper segments and the second is the outcome of joinning the lower segments.

Return type (*tuple*)

Examples

```
>>> from shapely.geometry import LineString
>>> from pybimstab.tools import upperLowerSegm, getPointAtX
>>> line = LineString([(0, 0), (5, 0)])
>>> x = 2.5
>>> point = getPointAtX(line, x)
>>> point.x, point.y
(2.5, 0.0)
```

tools.**extractSegment**(*line*, *distance1*, *distance2*)

Extracts a segment from a polyline (shapely LineString) given two distances measured from its starting point.

The following rule must be satisfied: *distance1* > *distance2*

Parameters

- **line** (*LineString*) – Any line or polyline from the `shapely` module.
- **distance1** (*int* or *float*) – The first distance measured from the starting pint of the line.
- **distance2** (*int* or *float*) – The second distance measured from the starting pint of the line.

Returns Two lines where each one is given in a (n, 2) array with the coordinates. The first one is the outcome of joinning the upper segments and the second is the outcome of joinning the lower segments.

Return type (*tuple*)

Examples

```
>>> from shapely.geometry import LineString
>>> from pybimstab.tools import upperLowerSegm, extractSegment
>>> line = LineString([(0, 0), (5, 0)])
>>> distance1, distance2 = 1, 3
>>> segment = extractSegment(line, distance1, distance2)
>>> list(segment.coords)
[(1.0, 0.0), (3.0, 0.0)]
```

2.10 watertable

Module for defining the classes related to the water table of a slope stability problem.

class watertable.**WaterTable**(*slopeCoords*, *watertabDepths*, *smoothFactor=0*)

Bases: object

Creates an instance of an object that defines the structure of the water table of a slope stability problem.

```
WaterTable(slopeCoords, watertabDepths, smoothFactor=0)
```

The water table is defined as tuples of points where the first value is the relative distance from the left most point of the slope and the second one is the relative depth from the terrain surface.

Some rules have to be followed: The distance equal to zero must be introduced; the distance equal to the horizontal length of the terrain surface must be introduced unless the last depth is zero; if the last input depth is equal to zero, the water table will continue with the surface shape.

slopeCoords

Coordinates of the slope which is supossed to be a closed polygon. It can be gotten from the method `defineBoundary` either from the classes `AnthropicSlope` or `NaturalSlope` (both in the module `SlopeGeometry`), however it works for any polygon.

Type (2, n) *numpy.ndarray*

watertabDepths

Relative coordinates to the slope surface of the polyline which defines the watertable. First row contains the horizontal relative distances from the left most point on the terrain surface and the second row contains the depths meassured from the terrain surface.

Type (2, n) *numpy.ndarray*

smoothFactor

Value to indicate the B-spline interpolation order of the smooter function. If is equal to zero, which is the default value, the surface will not be smoothed. It is suggested that the smooth factor be equal to 2 or 3 because higher values tend to lower the water table due to the smoothing.

Type *int*

Note: The class `WaterTable` requires numpy, matplotlib and shapely.

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.watertable import WaterTable
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
>>> watertabDepths = array([[0, 2, 5, 7, 12, 15],
>>>                         [2.5, 2.5, 3, 1.5, 0.5, 1]])
>>> watertable = WaterTable(slopeCoords=slope.coords,
>>>                         watertabDepths=watertabDepths,
>>>                         smoothFactor=1)
>>> watertable.__dict__.keys()
dict_keys(['slopeCoords', 'watertabDepths', 'smoothFactor', 'coords'])
```

defineStructre()

Method to define the structure of the water table

If the polyline which defines the water table intersects the terrain surface, it will force the water table keeps on the terrain and not above it.

Returns Absolute coordinates of the vertices of the polyline which defines the water table. First row contains the abcsises and the second row contains the ordinates.

Return type ((2, n) *numpy.ndarray*)

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.watertable import WaterTable
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
>>> watertabDepths = array([[0, 2, 5, 7, 12, 15],
>>>                         [2.5, 2.5, 3, 1.5, 0.5, 1]])
>>> watertable = WaterTable(slopeCoords=slope.coords,
>>>                         watertabDepths=watertabDepths,
>>>                         smoothFactor=1)
>>> watertable.defineStructre()
array([[ 0.        ,  0.20408163,  0.40816327,  0.6122449 ,...],
       [ 6.875     ,  6.875     ,  6.875     ,  6.875     ,...]])
```

plot()

Method for generating a graphic of the water table and the slope.

Returns object with the matplotlib structure of the plot. You might use it to save the figure for example.

Return type (*matplotlib.figure.Figure*)

Examples

```
>>> from numpy import array
>>> from pybimstab.slope import AnthropicSlope
>>> from pybimstab.watertable import WaterTable
>>> slope = AnthropicSlope(slopeHeight=7.5, slopeDip=[1, 1.5],
>>>                        crownDist=5, toeDist=5)
>>> watertabDepths = array([[0, 2, 5, 7, 12, 15],
>>>                         [2.5, 2.5, 3, 1.5, 0.5, 1]])
>>> watertable = WaterTable(slopeCoords=slope.coords,
>>>                         watertabDepths=watertabDepths,
>>>                         smoothFactor=0)
>>> fig = watertable.plot()
```

```
>>> watertable = WaterTable(slopeCoords=slope.coords,
>>>                         watertabDepths=watertabDepths,
>>>                         smoothFactor=3)
>>> watertable.plot()
```

```
>>> from numpy import array
>>> from pybimstab.slope import NaturalSlope
>>> from pybimstab.watertable import WaterTable
>>> terrainCoords = array(
>>>     [[-2.49, 0.1, 1.7, 3.89, 5.9, 8.12, 9.87, 13.29, 20.29,
>>>       21.43, 22.28, 23.48, 24.65, 25.17],
>>>      [18.16, 17.88, 17.28, 15.73, 14.31, 13.58, 13, 3.61, 3.61,
>>>       3.32, 2.71, 2.23, 1.21, 0.25]])
>>> slope = NaturalSlope(terrainCoords)
>>> watertabDepths = array([[0, 5, 10, 15, 20, 25, 27.66],
>>>                         [8, 7, 6, 3, 1, 1, 0.5]])
>>> watertable = WaterTable(slopeCoords=slope.coords,
>>>                         watertabDepths=watertabDepths,
>>>                         smoothFactor=3)
>>> fig = watertable.plot()
```

```
>>> from numpy import array
>>> from pybimstab.slope import NaturalSlope
>>> from pybimstab.watertable import WaterTable
>>> terrainCoords = array(
>>>     [[-2.49, 0.1, 1.7, 3.89, 5.9, 8.12, 9.87, 13.29, 20.29,
>>>       21.43, 22.28, 23.48, 24.65, 25.17],
>>>      [18.16, 17.88, 17.28, 15.73, 14.31, 13.58, 13, 3.61, 3.61,
>>>       3.32, 2.71, 2.23, 1.21, 0.25]])
>>> slope = NaturalSlope(terrainCoords)
>>> watertabDepths = array([[0, 5, 10, 15],
>>>                         [8, 7, 3, 0]])
>>> watertable = WaterTable(slopeCoords=slope.coords,
>>>                         watertabDepths=watertabDepths,
>>>                         smoothFactor=3)
>>> fig = watertable.plot()
```

CHAPTER 3

Use and Examples

The application software is able to evaluate both homogeneous and made of BIM slopes. Two examples of those cases are shown below:

3.1 Homogeneus slope without watertable

This example is the simplest one that can be executed with **pyBIMstab**:

```
# Example Case 1 - Fig. 9 (Fredlund & Krahn, 1977)
from pybimstab.slope import AnthropicSlope
from pybimstab.slipsurface import CircularSurface
from pybimstab.slices import MaterialParameters, Slices
from pybimstab.slopestabl import SlopeStabl
slope = AnthropicSlope(slopeHeight=40, slopeDip=[2, 1],
                       crownDist=60, toeDist=30, depth=20)
surface = CircularSurface(slopeCoords=slope.coords,
                          dist1=45.838, dist2=158.726,
                          radius=80)
material = MaterialParameters(cohesion=600, frictAngle=20,
                              unitWeight=120,
                              wtUnitWeight=62.4)
slices = Slices(
    material=material, slipSurfCoords=surface.coords,
    slopeCoords=slope.coords, numSlices=50,
    watertabCoords=None, bim=None)
stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0,
                          interSlcFunc=1)
fig = stabAnalysis.plot()
```

3.2 Slope made of BIM with watertable

This example is the most complex one that can be executed with **pyBIMstab**:

```
from numpy import array
from pybimstab.slope import NaturalSlope
from pybimstab.watertable import WaterTable
from pybimstab.bim import BlocksInMatrix
from pybimstab.slipsurface import CircularSurface, TortuousSurface
from pybimstab.slices import MaterialParameters, Slices
from pybimstab.slopestabl import SlopeStabl
terrainCoords = array(
    [[-2.49, 0.1, 1.7, 3.89, 5.9, 8.12, 9.87, 13.29, 20.29,
      21.43, 22.28, 23.48, 24.65, 25.17],
     [18.16, 17.88, 17.28, 15.73, 14.31, 13.58, 13, 3.61, 3.61,
      3.32, 2.71, 2.23, 1.21, 0.25]])
slope = NaturalSlope(terrainCoords)
bim = BlocksInMatrix(slopeCoords=slope.coords, blockProp=0.2,
                     tileSize=0.35, seed=3210)
watertabDepths = array([[0, 5, 10, 15],
                        [8, 7, 3, 0]])
watertable = WaterTable(slopeCoords=slope.coords,
                        watertabDepths=watertabDepths,
                        smoothFactor=3)
preferredPath = CircularSurface(
    slopeCoords=slope.coords, dist1=5, dist2=15.78, radius=20)
surface = TortuousSurface(
    bim, dist1=4, dist2=15.78, heuristic='euclidean',
    reverseLeft=False, reverseUp=False, smoothFactor=2,
    preferredPath=preferredPath.coords, prefPathFact=2)
material = MaterialParameters(
    cohesion=15, frictAngle=23, unitWeight=17,
    blocksUnitWeight=21, wtUnitWeight=9.8)
slices = Slices(
    material=material, slipSurfCoords=surface.coords,
    slopeCoords=slope.coords, numSlices=10,
    watertabCoords=watertable.coords, bim=bim)
stabAnalysis = SlopeStabl(slices, seedFS=1, Kh=0)
fig = stabAnalysis.plot()
```

CHAPTER 4

Authors

- Exneyder A. Montoya-Araque <eamontoyaa@gmail.com>
- Ludger O. Suarez-Burgoa <losuarezb@unal.edu.co>

CHAPTER 5

License

CHAPTER 6

History

6.1 0.1.0 (2018-07-15)

- First release on PyPI.

6.2 0.1.1 (2018-07-22)

- Solving some issues related to the definition of the slices structure.

6.3 0.1.2 (2018-08-04)

- Adjusting the interpolations in the convergence plot and appending a parameter to control the number of lambda values.

6.4 0.1.3 (2018-10-06)

- Addition of Bishop and Fellenius methods for LEM.
- Fixing some issues with convergences.
- Fixing minor issues.

6.5 0.1.4 (2019-10-13)

- Fixing minor issues.

6.6 0.1.5 (2019-10-15)

- Editing __init__.py file to import modules by means of an alias.

CHAPTER 7

References

D. G. Fredlund and J. Krahn. Comparison of slope stability methods of analysis. Canadian Geotechnical Journal, 14(3)(3):429–439, 1977.

GEO-SLOPE. Stability Modeling with SLOPE/W. An Engineering Methodology. GEO-SLOPE International Ltd, June 2015.

P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost path. IEEE Transactions of Systems Science and Cybernetics, ssc-4(2):100–107, 1968.

D. Y. Zhu, C. F. Lee, Q. H. Qian, and G. R. Chen. A concise algorithm for computing the factor of safety using the morgenstern–price method. IEEE ransactions of Systems Science and Cybernetics, 42:272–278, 2005.

CHAPTER 8

Links

- Documentation
- PyPI
- GitHub

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

CHAPTER 10

License and Copyright

Python Module Index

Index

X

Y