layermesh Documentation

Release 0.4.0

Adrian Croucher

Apr 04, 2024

CONTENTS

Welcome to the Layermesh user documentation.

CHAPTER

ONE

INTRODUCTION

1.1 What is Layermesh?

Layermesh is a Python library for creating and manipulating computational meshes with a layer/column structure, i.e. a (possibly unstructured) 2-D mesh projected down through a series of layers of constant thickness.

The uppermost layers of the mesh may be incomplete (i.e. do not contain cells for all columns), so that an irregular top surface can be used to represent e.g. topography.

The Layermesh library can be used to carry out a variety of actions on such meshes, including:

- creating meshes
- loading and saving from HDF5 files
- exporting to a variety of 3-D mesh formats (via the meshio library)
- fitting surface elevation data
- local refinement of the horizontal mesh
- optimization to improve horizontal mesh quality
- mesh searching, to locate particular cells, columns or layers
- 2-D layer and vertical slice plots (via Matplotlib)

1.2 Installation

Layermesh can be installed via `pip`, Python's package manager:

```
pip install layermesh
```

or if you don't have permissions for installing system-wide Python packages, you can just install it locally inside your own user account:

```
pip install --user layermesh
```

This will download and install Layermesh from the Python Package Index (PyPI).

1.3 Dependencies

Layermesh depends on several other Python libraries:

- `numpy`: Numerical Python
- `scipy`: Scientific Python
- `h5py`: Python interface for HDF5
- `meshio`: Python library for mesh file input/output
- `matplotlib`: Python plotting library

These will be installed automatically if not already present, if `pip` is used as above to install Layermesh.

1.4 Licensing

Layermesh is open-source software, released under the GNU Lesser General Public License (LGPL) version 3.

CHAPTER

TWO

GETTING STARTED

2.1 Importing Layermesh in a Python script

Before you can use Layermesh in a Python script, it must be imported (just like any other Python package). The package name is lowercase: `layermesh`.

The `layermesh` package contains several modules, the most important of which is the `mesh` module. There are several different ways a module can be imported from a Python library. Perhaps the simplest is to use the following syntax:

```
from layermesh import mesh
```

This imports only the `mesh` module from the `layermesh` package (which is typically all that is needed - the other Layermesh modules are just ones used by the `mesh` module). Commands from this module must be prefixed by the module name, `mesh`.

It is possible to import the module under a different name. For example:

```
from layermesh import mesh as lm
```

imports the mesh module and renames it to `lm`. Then, mesh commands would be prefixed by `lm` instead of `mesh`.

The same thing can also be achieved using:

```
import layermesh.mesh as lm
```

2.2 Layermesh classes

Layermesh provides the following main Python classes for representing meshes and mesh components.

- `mesh`: class for a layer/column mesh
- `node`: class for a 2-D horizontal mesh node
- `column`: class for a mesh column, defined by a list of `node` objects
- `layer`: class for a mesh layer, defined by its top and bottom elevations
- `cell`: class for a mesh cell at a particular `layer` and `column`

For full documentation of these classes, see the *Layermesh API*.

2.2.1 The mesh class

A mesh object represents an entire mesh.

It has list properties containing its nodes, columns, layers and cells. These are called node, column, layer and cell respectively, and their elements are all objects of the appropriate type.

2.2.2 The node class

A node object is defined mainly by its position property pos, a numpy array of length 2, representing its horizontal location. It also has a column property, a set of the columns the node belongs to.

2.2.3 The column class

A column object is defined mainly by its nodes, which are stored in its node property - a list of node objects. It also has a neighbour property, a set of the neighbouring columns (those which share a face).

A column object also has layer and cell list properties, containing the layers and cells in the column. Note that different columns may have different numbers of layers, as the upper layers in the mesh may be incomplete, to represent e.g. surface topography.

Columns also have geometric properties derived from their node positions, e.g. area and centroid, and a surface property, which is the elevation of the top of the column.

2.2.4 The layer class

A layer object is defined mainly by its top and bottom properties, which are scalars representing the top and bottom elevations of the layer.

A layer object also has column and cell list properties, containing the columns and cells in the layer, as well as a column_cell property, for locating layer cells by their column index. Note that different layers may have different numbers of columns, as the upper layers may be incomplete.

Each layer in a mesh has above and below properties, which are the layer objects above and below that layer, if they exist. If not (e.g. for the below property of the bottom layer), they have the value None.

Layers also have geometric properties derived from their top and bottom elevations, e.g. centre and thickness.

2.2.5 The cell class

A cell object is defined by its layer and column properties, which are the layer and column objects corresponding to the cell.

Cells have geometric properties such as volume and centroid. Other useful properties can be accessed via the column and layer properties. For example, for a cell object c, the horizontal area is given by c.column.area, and its vertical height is given by c.layer.thickness.

A cell also has a neighbour property, which is a set of its neighbouring cells, i.e. those with which it shares a face (either horizontal or vertical). The cell immediately above or below any cell can be found using its above and below properties. These return None if there is no cell respectively above or below that cell.

2.2.6 Index properties

Instances of the `node`, `column`, `layer` and `cell` classes all have an `index` property. This represents their index in the corresponding list in the mesh they belong to.

For example, for a column `col` which is part of a mesh `m`, `col.index` gives the index of `col` in the `m.column` list.

2.3 Creating rectangular meshes

A simple rectangular `mesh` object can be created by using the `rectangular` parameter. This is a list or tuple of the mesh spacings in each coordinate direction. Each mesh spacing specification is itself a list, tuple or array of spacings.

For example:

```
import layermesh.mesh as lm
m = lm.mesh(rectangular = ([1000]*10, [800]*12, [100]*8))
```

creates a simple regular rectangular 10×12×8 cell mesh, with constant mesh spacings in the x-, y- and z-directions of 1000, 800 and 100 respectively.

Irregular rectangular meshes can be created by passing non-uniform mesh spacings in in the `rectangular` parameter. For example:

```
import layermesh.mesh as lm
import numpy as np

dx = np.arange(1000, 7000, 1000)
dy = dx
dz = np.arange(10, 60, 10)
m = lm.mesh(rectangular = [dx, dy, dz])
```

creates an irregular rectangular mesh with equal spacings in the x- and y-directions ranging from 1000 to 6000, and with layer thicknesses ranging from 10 at the top to 50 at the bottom.

2.4 Mesh properties

Some of the other useful properties of a `mesh` object, besides its main list properties `node`, `column`, `layer` and `cell`, are:

- `num_nodes`, `num_columns`, `num_layers`, `num_cells`: numbers of nodes, columns, layers and cells in the mesh
- `area`: horizontal area occupied by the mesh
- `centre`: horizontal centre of the mesh
- `bounds`: horizontal bounding rectangle around the mesh
- `volume`: total volume occupied by the mesh cells
- `surface_cells`: list of cells at the top surface of the mesh

2.5 Reading and writing HDF5 mesh files

A Layermesh `mesh` object can be written to an HDF5 file using its `write()` method, which takes a filename as its parameter, e.g.:

```
msh.write('mymesh.h5')
```

writes the `mesh` object `msh` to the file "mymesh.h5".

Similarly, a `mesh` object can be read in from file by passing in a filename when creating it:

```
import layermesh.mesh as lm
msh = lm.mesh('mymesh.h5')
```

creates a new `mesh` object called `msh` and reads its contents from the file "mymesh.h5".

Layermesh HDF5 files have a simple structure with four groups:

- `cell`: one integer scalar `type_sort` dataset containing the value of the mesh `cell_type_sort` property
- `layer`: one rank-1 array float `elevation` dataset containing, in order, the `top` property of each mesh layer, and finally the `bottom` property of the last (bottom) layer
- `node`: one rank-2 array float `position` dataset containing the `pos` property of each node (horizontal position) in the mesh `node` list
- `column`: an rank-2 integer `node` dataset containing the index of each node in the column, for each column in the mesh `column` list (columns with fewer nodes padded out with -1 values) ; and also a rank-1 integer `num_layers` dataset containing the number of layers for each column

2.6 Fitting surface elevation data

Layermesh meshes may have incomplete upper layers (i.e. different columns may have different numbers of layers) to represent e.g. surface topography. The surface of the mesh can be specified by fitting arbitrary scattered (x, y, z) data, using the `mesh` `fit_surface()` method.

This method uses least-squares finite element fitting with piecewise constant elements to determine an appropriate surface elevation for each column. The number of layers in the column is then determined by taking this fitted elevation and choosing the nearest layer boundary as the top surface of the column.

On its own, however, this algorithm will fail if the dataset is sparse and there are columns which do not contain any data points. To overcome this (and also to help overcome problems with noisy data) an additional smoothing term is introduced to the least-squares fitting process. This term is simply the sum of squares of the differences in elevation across the faces between columns. This term is weighted by a `smoothing` parameter (with default value 0.01) which may be passed into the `fit_surface()` method.

For example:

```
import layermesh.mesh as lm
import numpy as np

m = lm.mesh(rectangular = ([1000]*10, [800]*12, [100]*8))
surf = np.loadtxt('surface.txt')
m.fit_surface(surf, smoothing = 0.02)
```

creates a simple rectangular mesh, loads surface elevation data from a text file containing (x, y, z) data on each line, and fits the mesh surface to the data using a smoothing parameter of 0.02.

Generally only a small value of the smoothing parameter is needed to overcome problems with sparse data. Its value can be increased if the dataset is noisy and there are large gradients in the fitted surface.

It is also possible to fit data over only some of the mesh columns, rather than all of them (the default). To do this, the `columns` parameter is used, which takes a tuple or list of columns to be fitted:

```
cols = m.find([(0,0), (5000, 5000)])
m.fit_surface(surf, columns = cols, smoothing = 0.02)
```

Here surface fitting is carried out for all columns with centroids within a rectangle with bottom left coordinates at the origin and top right coordinates (5000, 5000). (For more information on how to find particular mesh columns or other mesh components using the `find()` method, see *Searching*.)

2.7 Exporting to other formats

The purpose of using Layermesh is usually to create a computational mesh which can be used by other software (such as a flow simulator or 3-D visualisation package). This involves expanding the layer/column structure of a Layermesh mesh into a full 3-D mesh, which can then be exported to a mesh format which other software can read.

This can be done by using the `mesh export()` method, which takes a filename as its parameter. The meshio library is used to write the mesh, so the mesh can be exported to any mesh format that `meshio` understands (ExodusII, GMSH, VTU, XDMF, H5M and more). The desired format is determined from the filename extension. (Alternatively, it can be explicitly specified using the `fmt` parameter.)

For example:

```
import layermesh.mesh as lm
m = lm.mesh('mymesh.h5')
m.export('mymesh.vtu')
m.export('mymesh.msh')
```

reads a mesh from a Layermesh HDF5 file and exports it twice, first to a VTU file for 3-D visualisation using e.g. Paraview, and then to GMSH `*.msh` format.

2.8 Searching

Searching for particular Layermesh mesh components (e.g. cells, columns or layers) can be carried out using the `mesh find()` method. This method can be used in several different ways, depending on what kind of parameters it is given.

2.8.1 Searching for cells

Cells can be found based on their 3-D position, or via a user-specified function to select cells with particular attributes.

Searching by position

Passing a 3-D point as the parameter to the `mesh` `find()` method will return the cell containing that point. The point can be specified as a tuple, list or `numpy` array.

For example:

```
c = m.find((1200, 3450, -400))
```

finds the cell in the mesh `m` containing the point (1200, 3450, -400) and stores it in the variable `c` (a `cell` object).

When used in this way, the `find()` method first determines the layer containing the elevation of the point, and then searches that layer for the appropriate column, using a quadtree search. If the mesh does not contain the specified point, the `find()` method will return `None`.

Sometimes it may be more convenient to return the index of the cell, rather than the cell object. This can be done by setting the `indices` parameter to `True`:

```
i = m.find((1200, 3450, -400), indices = True)
```

In this case, `i` is an integer representing the cell index. (However, `None` will still be returned if the point is not inside the mesh.)

Searching using a function

It is also possible to use the `find()` method to search for cells with particular attributes defined using a function. The function, typically user-defined, must take a cell as its argument and return a Boolean (`True` or `False`). The `find()` method will then return a list of all mesh cells for which the function is `True`.

For example, supposing we wish to find all mesh cells with volume greater than 1000 and centre elevation below -600. To do this, we can define a suitable function, and pass it to `find()`:

```
def f(c):
  return c.volume > 1000 and c.layer.centre < -600

cells = m.find(f)
```

Note that using a function to find cells in this way may not be efficient for large meshes, as it involves a full search over all mesh cells.

2.8.2 Searching for columns

Searching by position

Passing a 2-D point (tuple, list or array) into the `mesh` `find()` method will return the column containing that horizontal position, for example:

```
p = np.array([3100, 4410])
col = m.find(p)
```

returns the column object containing the point (3100, 4410) (represented here by the `numpy` array `p`) and stores it in the variable `col`. As for cells, setting the `indices` parameter to `True` means that column indices can be returned instead of column objects. In either case, `None` is returned if the point is outside the mesh.

Searching for columns inside a polygon

Passing a polygon of 2-D points into the `mesh find()` method will return a list of all columns with centroids inside that polygon.

Here, a polygon is represented by a tuple, list or array of 2-D points (each one a tuple, list or array of length 2). To search inside a rectangle, only the bottom left and top right corner points need be specified (any polygon with only two points will be interpreted as a rectangle).

For example:

```
cols = m.find([(0,0), (3500, 4200)])
```

finds all columns in the rectangle with bottom left corner at the origin and top right corner at (3500, 4200).

```
cols = m.find([(0,0), (3500, 1100), (900, 5100)])
```

finds all columns in the triangle with corners at the specified points. Polygons may have any number of points.

2.8.3 Searching for layers

Passing a scalar into the `mesh find()` method will return the layer containing the specified elevation, e.g.:

```
lay = m.find(-2400)
```

returns the layer containing the elevation -2400. Again, the `indices` parameter can be used to return layer indices rather than layer objects, and `None` is always returned if the elevation is outside the mesh.

2.8.4 Searching within columns or layers

Mesh columns and layers also have a `find()` method, which works very similarly to that of the mesh itself. Passing a 3-D point as parameter will return the cell containing that point (or `None` if it is outside), e.g.:

```
c = m.layer[-1].find((3450, 1200, -340))
```

finds the cell in the bottom layer (index -1) of the mesh containing the point (3450, 1200, -340).

```
c = m.column[12].find((3450, 1200, -340))
```

searches column 12 in the mesh for the same 3-D point.

Passing a 2-D point will return the column containing that point. If a column is being searched, the result will be either the column itself, or `None`. For example:

```
col = m.layer[2].find((230, 345))
```

finds the column in mesh layer 2 containing the point (230, 345). Note that the search results can be different in different layers, because not all of them necessarily contain the same columns (if there are incomplete layers at the surface).

```
if m.column[12].find((230, 345)):
  # do something
```

uses `find()` in a conditional to execute some code if the horizontal point (230, 345) is inside column 12 of the mesh.

Passing a scalar will return the layer containing that elevation. If a layer is being searched, either the layer itself or `None` will be returned. For example:

```
lay = m.column[12].find(-100)
```

returns the layer in column 12 containing the elevation -100.

```
if m.layer[-1].find(-3000):
  # do something
```

executes a conditional statement if the bottom mesh layer contains the elevation -3000.

Cells in columns and layers can also be found using a function, in exactly the same way this is done for a mesh.

2.8.5 Searching within cells

It is also possible to search within a cell. This amounts to determining if the cell contains the specified 3-D point, 2-D horizontal position or scalar elevation. If it does, the cell, column or layer itself is returned. If it doesn't, `None` is returned.

For example:

```
if m.cell[2].find((230, 540, -250)):
  # do something
```

executes a conditional if cell 2 in the mesh contains the 3-D point (230, 540, -250).

Passing a polygon into a cell's `find()` method will return the cell's column if its centroid is inside the polygon (or `None` otherwise).

2.9 Horizontal refinement

Horizontal mesh refinement (i.e. refinement of the column structure) can be carried out using the `mesh` `refine()` method. It is possible to refine only selected parts of the mesh using the `columns` parameter, which is a set, tuple or list of `column` objects to be refined.

For example:

```
cols = m.find((0, 0), (4000, 5000))
m.refine(cols)
```

will refine all mesh columns within the rectangle with lower left corner at the origin and upper right corner at (4000, 5000).

The selected columns are replaced by four refined columns (the edges of the original columns being subdivided in two). Triangular columns are added around the edge of the refinement area to make the transition from coarse to fine columns.

Note that the triangular transition columns created by `refine()` may not necessarily have desirable mesh quality statistics (e.g. aspect ratios or face orthogonality). Hence it is often necessary to follow the `refine()` command with a call to the `optimize()` method (see *Optimizing a mesh*), in order to regain acceptable mesh quality in the transition region.

2.10 Optimizing a mesh

The accuracy of the results generated from a simulation on a computational mesh is dependent (in part) on the quality of the mesh. The way mesh quality is measured depends on the type of simulation being carried out.

For some types of finite element simulation, for example, elements should ideally have small aspect ratios and low skewness. For some types of finite volume simulations, on the other hand, the orthogonality of the mesh faces is important for accurate results.

Layermesh `mesh` objects have an `optimize()` method for improving mesh quality. This method uses a least-squares minimization technique to move the mesh node positions in such a way as to maximize specified mesh quality measures.

Any weighted combination of aspect ratio, skewness and face orthogonality may be used in the optimization. This is specified via the `weight` parameter, which is a dictionary with up to three keys: "aspect", "skewness" and "orthogonal". The values assigned to these keys are the desired relative weights of the corresponding mesh quality measures in the optimization.

In general it is not advisable to attempt to optimize the entire mesh at once. This gives an optimization with too many degrees of freedom, resulting in long processing times and a greater chance of either non-convergence or convergence to a nonsensical result. It is better to focus the optimization on those areas of the mesh that are known to need improvement. For example, if the mesh has had horizontal local refinement (see *Horizontal refinement*), the triangular transition columns may have low quality, in which case the optimization can be concentrated on those columns.

The optimization can be limited to either specified nodes, or specified columns. In the latter case, all nodes in the specified columns are selected for optimization.

For example:

```
triangles = m.type_columns(3)
m.optimize(columns = triangles)
```

optimizes all nodes in triangular columns, using the default weighting, which gives face orthogonality a weight of 1 and other measures zero (i.e. only face orthogonality is optimized).

```
cols = m.find([(0, 0), (4000, 5000)])
m.optimize(columns = cols, weight = {'aspect': 0.75, 'skewness': 0.25})
```

Here all nodes in columns within a specified rectangle are optimized, giving 75% weight to aspect ratio and 25% to skewness in the optimization. (Orthogonality is not specified, so it is given zero weight.)

2.11 Creating 2-D plots

Layermesh can be used to create 2-D plots of the mesh, with cells optionally labelled and/or shaded with values (e.g. simulation results).

Layermesh `mesh` objects have two methods for creating plots:

- the `layer_plot()` method creates a plot over a specified mesh layer
- the `slice_plot()` method creates a plot over a specified vertical slice through the mesh

In either case, the Matplotlib library is used to create the plot, which can be either viewed directly on the display or saved to an image file.

2.11.1 Layer plots

The `mesh` `layer_plot()` method takes as its first parameter the layer to be plotted - either a `layer` object, or an integer mesh layer index. Alternatively, an elevation can be specified via the `elevation` parameter, which will then be used to determine the appropriate layer. If neither the layer or an elevation is specified, then the bottom layer is plotted.

Examples:

```
m.layer_plot() # plot bottom layer

lay = m.layer[2]
m.layer_plot(lay) # plot layer 2

m.layer_plot(elevation = -1350) # plot layer containing elevation -1350
```

2.11.2 Slice plots

The `mesh` `slice_plot()` method takes as its first parameter the line defining the slice to be plotted. This can be either:

- a string "x" or "y" to plot through the mesh centre along the *x*- or *y*-axes
- a number representing an angle (in degrees clockwise from the *y*-axis) to plot through the mesh centre on that angle
- a tuple, list or array of two 2-D points representing the end-points of the line

Examples:

```
m.slice_plot() # plot through centre along x-axis

m.slice_plot('y') # plot through centre along y-axis

m.slice_plot(45) # plot through centre at 45 degrees from y-axis

line = [(0,0), (3000, 4000)]
m.slice_plot(line) # plot along specified line
```

2.11.3 Plotting values over the mesh

Both `layer_plot()` and `slice_plot()` take an optional `value` parameter, which is a tuple, list or rank-1 array of values to plot over the mesh. The length of the `value` parameter should be equal to the number of mesh cells. For example:

```
T = np.loadtxt('temperatures.txt')
m.layer_plot(elevation = -50, value = T)
```

loads an array of values from a text file and plots them over the layer at elevation -50.

When a value is plotted, a colourbar scale is drawn next to the plot. The optional `value_label` and `value_unit` parameters can be used to produce the name of the quantity being plotted on the colourbar, together with its units, e.g.:

```
m.layer_plot(elevation = -50, value = T,
  value_label = 'Temperature', value_unit = 'deg C')
```

2.11.4 Plotting labels

The `layer_plot()` and `slice_plot()` methods also have an optional `label` parameter, if labels are to be drawn at the centre of each cell in the plot.

The `label` parameter is a string and can be either:

- "cell": label cells with cell indices
- "value": label cells with numerical values, taken from the `value` parameter
- "column" (`layer_plot()` only): label cells with column indices

Examples:

```
m.slice_plot('x', label = 'cell') # plot along x-axis, labelling cell indices

m.layer_plot(10, label = 'column') # plot layer 10, labelling column indices

m.slice_plot('y', value = T, label = 'value') # plot and label T along y-axis
```

2.11.5 Plot output

By default, the `layer_plot()` and `slice_plot()` methods plot directly to the display, so a plot will appear immediately after the method is called.

It is also possible to plot to a Matplotlib `axes` object instead, via the `axes` parameter of the `layer_plot()` and `slice_plot()` methods. This can be useful for e.g.:

- putting multiple plots on one page
- superimposing other things on the plot
- saving the output to an image file

For example:

```
import layermesh.mesh as lm
import numpy as np
import matplotlib.pyplot as plt

m = lm.mesh('mymesh.h5')
P = np.loadtxt('pressures.txt')
T = np.loadtxt('temperatures.txt')

fig = plt.figure()

ax = fig.add_subplot(2, 1, 1)
m.slice_plot('x', axes = ax, value = P,
  value_label = 'Pressure', value_unit = 'bar')

ax = fig.add_subplot(2, 1, 2)
m.slice_plot('x', axes = ax, value = T,
  value_label = 'Temperature', value_unit = 'deg C')

plt.suptitle('Pressure and temperature plots along x-axis')
plt.savefig('plots.png')
```

Here a mesh is loaded from an HDF5 file, along with the datasets `P` and `T` which are loaded from text files. A Matplotlib figure is created, and within it, axes for two subplots. These are used to call `slice_plot()` twice, to plot `P` and `T` along an *x*-axis slice.

Finally, the plot is given a title and the output saved to an image file.

If the `axes` parameter is passed to `layer_plot()` or `slice_plot()`, nothing will appear on the display when the method is called. In the above example the plot could be shown by adding:

```
plt.show()
```

CHAPTER

THREE

LAYERMESH API

3.1 layermesh package

3.1.1 Submodules

layermesh.geometry module

Geometry calculations.

layermesh.geometry.**bounds_of_points**(*points*)

Returns bounding box around the specified tuple, list, array or set of points, each one a tuple, list or array of length 2.

layermesh.geometry.**in_polygon**(*pos*, *polygon*)

Tests if the point *pos* (a tuple, list or array of length 2) a lies within a given polygon (a tuple or list of points, each itself a tuple, list or array of length 2).

layermesh.geometry.**in_rectangle**(*pos*, *rect*)

Tests if the point *pos* lies in an axis-aligned rectangle, defined as a two-element tuple or list of points [bottom left, top right], each itself a tuple, list or array of length 2.

layermesh.geometry.**line_intersects_rectangle**(*rect*, *line*)

layermesh.geometry.**line_polygon_intersections**(*polygon*, *line*, *bound_line=(True, True)*, *indices=False*)

Returns a list of the intersection points at which a line crosses a polygon. The list is sorted by distance from the start of the line. The parameter bound_line controls whether to limit intersections between the line's start and end points. If indices is True, also return polygon side indices of intersections.

layermesh.geometry.**line_projection**(*a*, *line*, *return_xi=False*)

Finds projection of point *a* onto a *line* (defined by two points, each a tuple, list or array of length 2). Optionally returns the non-dimensional distance xi between the line start and end.

layermesh.geometry.**point_line_distance**(*a*, *line*)

Finds the distance between point *a* and a line.

layermesh.geometry.**polygon_area**(*polygon*)

Calculates the (unsigned) area of an arbitrary polygon (a tuple, list or array of points, each one a tuple, list or array of length 2).

layermesh.geometry.**polygon_boundary**(*this*, *other*, *polygon*)

Returns point on a line between vector this and other and also on the boundary of the polygon.

layermesh.geometry.**polygon_centroid**(*polygon*)

Calculates the centroid of an arbitrary polygon (a tuple, list or array of points, each one a tuple, list or array of length 2).

layermesh.geometry.**polyline_line_distance**(*polyline*, *line*)

Returns minimum distance between a polyline and a line.

layermesh.geometry.**polyline_polygon_intersections**(*polygon*, *polyline*)

Returns a list of intersection points at which a polyline (a tuple, list or array of points, each one a tuple, list or array of length 2) crosses a polygon.

layermesh.geometry.**rect_to_poly**(*rect*)

Converts a rectangle to a polygon.

layermesh.geometry.**rectangles_intersect**(*rect1*, *rect2*)

Returns *True* if two rectangles intersect.

layermesh.geometry.**rotation**(*angle*, *centre=None*)

Returns 2-by-2 matrix A and vector b representing a rotation of the specified angle (degrees clockwise) about the specified centre (or the origin if no centre is specified). The rotation of a point p is then given by Ap + b.

layermesh.geometry.**simplify_polygon**(*polygon*, *tolerance=1e-06*)

Simplifies a polygon by deleting colinear points. The tolerance for detecting colinearity of points can optionally be specified.

layermesh.geometry.**sub_rectangles**(*rect*)

Returns the sub-rectangles formed by subdividing the given rectangle evenly in four.

layermesh.geometry.**vector_heading**(*p*)

Returns heading angle of a point p (tuple, list or array of length 2), in radians clockwise from the y-axis ('north').

layermesh.mesh module

Layered computational meshes.

class layermesh.mesh.**cell**(*lay*, *col*, *index=None*)

Bases: object

Mesh cell. On creation, the layer and column defining the cell (and optionally the cell index) are specified.

property above

Cell above the current cell, or None if there is no cell above it.

property below

Cell below the current cell, or None if there is no cell below it.

property centre

Centroid of cell.

property centroid

Centroid of cell.

column

Cell column object.

find(*match*, *indices=False*)

Returns cell, column or layer satisfying the criterion *match*.

The *match* parameter can be:

- a **function** taking a cell and returning a Boolean: the cell is returned if it matches, otherwise *None*
- a **scalar**: *match* is interpreted as an **elevation** and the cell layer is returned if the elevation is inside it
- a **2-D point** (tuple, list or array of length 2): *match* is interpreted as a **horizontal position**, and the cell column is returned if the position is inside it
- a **polygon** (tuple, list or array of 2-D points): the cell column is returned if the cell column centroid is inside the polygon
- a **3-D point** (tuple, list or array of length 3): the cell is returned if the point is inside it

If indices is *True*, the cell, column or layer index is returned rather than the cell, column or layer itself.

In each case, *None* is returned if there is no match.

index

Index of the cell in the mesh.

layer

Cell layer object.

property neighbour

Set of neighbouring cells in the mesh, i.e. those that share a common face.

property num_neighbours

Number of neighbouring cells in the mesh, i.e. those that share a common face.

property num_nodes

Number of nodes in the cell (at both top and bottom of layer).

property surface

True if the cell is at the surface of the mesh, *False* otherwise.

property volume

Volume of cell.

class layermesh.mesh.**column**(*node*, *index=None*)

Bases: _layered_object

Mesh column. On creation, the column's nodes (and optionally index) are specified.

property angle_ratio

Angle ratio, defined as the ratio of the largest interior angle to the smallest interior angle.

This can be used as a measure of the skewness of the column, with values near 1 being less skewed.

property area

Area of column.

property bounding_box

Horizontal bounding box of column.

cell

List of cells in the column.

property centre

Column centroid.

property centroid

Column centroid.

property face_length

Array of lengths of the column faces.

property face_length_ratio

Face length ratio, defined as the ratio of the longest face length to the shortest face length (a generalisation of the aspect ratio for quadrilateral columns).

find(*match*, *indices=False*, *sort=False*)

Returns cells, columns or layers satisfying the criterion *match*.

The *match* parameter can be:

- a **function** taking a cell and returning a Boolean: a list of matching cells is returned
- a **scalar**: *match* is interpreted as an **elevation**, and the layer containing it is returned
- a **2-D point** (tuple, list or array of length 2): *match* is interpreted as a **horizontal position**, and self is returned if it contains the point
- a **polygon** (tuple, list or array of 2-D points): self is returned if its centroid is inside the polygon
- a **3-D point** (tuple, list or array of length 3): *match* is interpreted as a **3-D position**, and the cell containing it is returned

If indices is *True*, the cell, column or layer indices are returned rather than the cells, columns or layers themselves.

If *sort* is *True*, then lists of results are sorted by index.

If no match is found, then *None* is returned, except when the expected result is a list, in which case an empty list is returned.

index

Integer containing the column's index in the mesh.

property interior_angle

Array of interior angles for each node in the column.

layer

List of layers in the column.

neighbour

Set containing the neighbouring columns (those that share a face).

node

List of the node objects in the column.

property num_cells

Number of cells in the column.

property num_layers

Number of layers in the column.

property num_neighbours

Number of neighbouring columns (those that share a face).

property num_nodes

Number of nodes in the column.

property polygon

Polygon (list of arrays of length 2) formed by column node positions.

set_layers(*layers*, *num_layers*)

Sets column layers to be the last *num_layers* layers from the specified list.

set_surface(*layers*, *surface=None*)

Sets column layers from the given list, according to the specified surface elevation.

If *surface* = *None*, then the column is assigned all layers in the list. Otherwise, it is assigned all layers with centres below the specified surface elevation.

property side_neighbour

List of neighbouring columns corresponding to each column side (or None if the column side is on a boundary).

property surface

Surface elevation of the column, given by the top elevation of its uppermost layer. (This property is read-only: use **set_surface()** or **set_layers()** to set the layers in the column.)

translate(*shift*)

Translates column horizontally by the specified shift array (a tuple, list or array of length 2).

property volume

Column volume.

class layermesh.mesh.**column_face**(*column*)

Bases: object

Face between two columns. On creation, the two columns on either side of the face are specified.

property angle_cosine

Cosine of angle between the face and the line joining the column centroids on either side.

This can be used to measure the orthogonality of the face: orthogonal faces have angle cosine zero.

column

List or tuple of column objects on either side of the face.

node

List of node objects at either end of the face.

class layermesh.mesh.**layer**(*bottom*, *top*, *index=None*)

Bases: object

Mesh layer. On creation, the bottom and top elevations of the layer (and optionally the layer index) are specified.

above

Layer above this one, if it exists, otherwise None.

property area

Horizontal area of layer.

below

Layer below this one, if it exists, otherwise None.

`bottom`

Bottom elevation of the layer.

`cell`

List of cells in the layer.

`property centre`

Elevation of layer centre.

`column`

List of columns in the layer.

`column_cell`

Dictionary of cells, keyed by column indices.

`find`(*match*, *indices=False*, *sort=False*)

Returns cells, columns or layer satifying the criterion *match*.

The *match* parameter can be:

- a **function** taking a cell and returning a Boolean: a list of matching cells is returned
- a **scalar**: *match* is interpreted as an **elevation**, and the layer is returned if the elevation is inside it
- a **2-D point** (tuple, list or array of length 2): *match* is interpreted as a **horizontal position**, and the column containing it is returned
- a **polygon** (tuple, list or array of 2-D points): a list of columns inside the polygon are returned
- a **3-D point** (tuple, list or array of length 3): *match* is interpreted as a **3-D position**, and the cell containing it is returned

If indices is *True*, the cell, column or layer indices are returned rather than the cells, columns or layer themselves.

If *sort* is *True*, then lists of results are sorted by index.

If no match is found, then *None* is returned, except when the expected result is a list, in which case an empty list is returned.

`property horizontal_bounds`

Horizontal bounding box for layer (list of two arrays of length 2, representing the bottom left and top right corner coordinates of the bounding box).

`index`

Layer index in the mesh (numbered from top down).

`property node`

Set of nodes in the layer.

`property num_cells`

Number of cells in the layer.

`property num_columns`

Number of columns in the layer.

`property quadtree`

Quadtree object for column searching within the layer.

`property thickness`

Vertical thickness of layer.

top

Top elevation of the layer.

translate(*shift*)

Translates layer by specified 3-D shift (a tuple, list or array of length 3).

property volume

Volume of layer.

class layermesh.mesh.**mesh**(*filename=None*, ***kwargs*)

Bases: _layered_object

A mesh can be created either by reading it from a file, or via other parameters.

If *filename* is specified, the mesh is read from the given HDF5 file.

Otherwise, a rectangular mesh can be created using the *rectangular* parameter. Mesh spacings in the three coordinate directions are specified via tuples, lists or arrays of spacings. The *rectangular* parameter is itself a tuple or list of three of these mesh spacing specifications.

The surface elevations can be specified using the *surface* parameter. This can be either a dictionary of pairs of column indices and corresponding surface elevations, or a tuple, list or array of surface elevations for all columns. If *None* is specified (the default) then all column surfaces will be set to the top of the uppermost layer.

By default, mesh cells are ordered first by cell type (number of nodes, in decreasing order), then layer and finally by column within each layer, from the top to bottom of the mesh. The sorting of cell types can be reversed or disabled by setting the *cell_type_sort* parameter: a value of 1 sorts cells in increasing type order, and a value of zero disables cell type sorting.

add_column(*col*)

Adds a column to the mesh.

add_layer(*lay*)

Adds a layer to the mesh.

add_node(*n*)

Adds a horizontal node to the mesh.

property area

Horizontal area of the mesh.

property boundary_nodes

Set of nodes on the boundary of the mesh.

property bounds

Horizontal bounding box for the mesh.

cell

List of cell objects in the mesh.

cell_type_sort

Integer controlling sorting of cells by type. A value of -1 (the default) gives cells sorted by decreasing type (number of nodes). A value of 1 gives cells sorted by increasing type, while a value of zero disables cell type sorting.

property centre

Horizontal centre of the mesh (an array of length 2), approximated by an area-weighted average of column centres.

column

List of column objects in the mesh.

column_faces(*columns=None*)

Returns a list of the column faces between the specified columns. A list of the columns may be optionally specified, otherwise all columns will be included.

column_in_layer(*col*, *lay*)

Returns *True* if column is in the specified layer, or *False* otherwise.

column_track(*line*)

Returns a list of tuples of (column,entrypoint,exitpoint) representing the horizontal track traversed by the specified line through the grid. Line is a tuple of two 2D arrays. The resulting list is ordered by distance from the start of the line.

delete_column(*col*)

Deletes the specified column object from the mesh.

export(*filename*, *fmt=None*)

Exports 3-D mesh using meshio, to file with the specified name. If the format is not specified via the fmt parameter, it is determined from the filename extension.

find(*match*, *indices=False*, *sort=False*)

Returns cells, columns or layers satisfying the criterion *match*.

The *match* parameter can be:

- a **function** taking a cell and returning a Boolean: a list of matching cells is returned
- a **scalar**: *match* is interpreted as an **elevation**, and the layer containing it is returned
- a **2-D point** (tuple, list or array of length 2): *match* is interpreted as a **horizontal position**, and the mesh column containing it is returned
- a **polygon** (tuple, list or array of 2-D points): a list of mesh columns inside the polygon are returned
- a **3-D point** (tuple, list or array of length 3): *match* is interpreted as a **3-D position**, and the mesh cell containing it is returned

If indices is *True*, the cell, column or layer indices are returned rather than the cells, columns or layers themselves.

If *sort* is *True*, then lists of results are sorted by index.

If no match is found, then *None* is returned, except when the expected result is a list, in which case an empty list is returned.

fit_data_to_columns(*data*, *columns=None*, *smoothing=0.01*)

Fits scattered data to the columns of the mesh, using smoothed piecewise constant least-squares fitting.

The data should be in the form of a 3-column array with x,y,z data in each row. Fitting can be carried out over a subset of the mesh columns by specifying a tuple or list of columns.

Increasing the smoothing parameter will decrease gradients between columns, and a non-zero value must be used to obtain a solution if any columns contain no data.

fit_surface(*data*, *columns=None*, *smoothing=0.01*)

Fits surface elevation data to determine the number of layers in each column.

The *data* should be in the form of a 3-column array with x,y,z data in each row. Fitting can be carried out over a subset of the mesh columns by specifying a tuple or list of columns.

Increasing the smoothing parameter will decrease gradients between columns, and a non-zero value must be used to obtain a solution if any columns contain no data.

identify_column_neighbours()

Identifies neighbours for each column.

layer

List of layer objects in the mesh.

layer_plot(*lay=-1*, ***kwargs*)

Creates a 2-D Matplotlib plot of the mesh at a specified layer. The *lay* parameter can be either a layer object or a layer index.

Other optional parameters:

- *aspect*: the aspect ratio of the axes (default *'equal'*).
- *axes*: a Matplotlib axes object on which to draw the plot. If not specified, then a new axes object will be created internally.
- *colourmap*: a Matplotlib colourmap object for shading the plot according to the *value* array (default *None*).
- *elevation*: used to specify an elevation instead of a layer.
- *label*: a string (or *None*, the default) specifying what labels are to be drawn at the centre of each column. Possible values are *'column'* (to label with column indices), *'cell'* (to label cell indices) or *'value'* (to label with the *value* array).
- *label_format*: format string for the labels (default *'%g'*).
- *label_colour*: the colour of the labels (default *'black'*).
- *linecolour*: the colour of the mesh grid (default *'black'*).
- *linewidth*: the line width of the mesh (default *0.2*).
- *value*: a tuple, list or array of values to plot over the mesh, of length equal to the number of cells in the mesh.
- *xlabel*: label string for the plot *x*-axis (default *'x'*).
- *ylabel*: label string for the plot *y*-axis (default *'y'*).

property meshio_points_cells

Lists of 3-D points and cells suitable for mesh input/output using meshio library.

node

List of node objects in the mesh.

nodes_in_columns(*columns*)

Returns a set of nodes in the specified columns.

property num_cells

Number of 3-D cells in the mesh.

property num_columns

Number of columns in the mesh.

property num_layers

Number of layers in the mesh.

property num_nodes

Number of 2-D nodes in the mesh.

optimize(*nodes=None*, *columns=None*, *weight={'orthogonal': 1}*)

Adjusts horizontal positions of specified nodes to optimize the mesh. If no nodes are specified, all node positions are optimized. If columns are specified, the positions of nodes in those columns are optimized.

Three types of optimization are offered, with their relative importance in the optimization specified via the *weight* dictionary parameter. This can contain up to three keys:

- 'orthogonal': the orthogonality of the mesh faces
- 'skewness': the skewness of the columns
- 'aspect': the aspect ratio of the columns

Omitting any of these keys from the *weight* parameter will give them zero weight. Weights need not sum to 1: only their relative magnitudes matter.

The optimization is carried out by using the `leastsq()` function from the `scipy.optimize` module to minimize an objective function formed from a weighted combination of the mesh quality measures above.

read(*filename*)

Reads mesh from HDF5 file.

refine(*columns=None*)

Refines selected columns in the mesh. If no columns are specified, then all columns are refined. The *columns* parameter can be a set, tuple or list of column objects.

Each selected column is divided into four new columns. Triangular transition columns are added around the edge of the refinement area as needed.

Note that the selected columns must be either triangular or quadrilateral (columns with more than four edges cannot be refined).

Mesh refinement will generally alter the indexing of the mesh nodes, columns and cells, even those not within the refinement area. Hence, it should not be assumed, for example, that columns outside the refinement area will retain their original indices after the refinement.

Based on the mulgrid refine() method in PyTOUGH.

rotate(*angle*, *centre=None*)

Rotates the mesh horizontally by the specified angle (degrees clockwise). If no centre is specified, the mesh is rotated about its own centre.

set_column_layers(*num_layers*)

Sets column layers from dictionary (keyed by column indices) or list/array of layer counts for each column.

set_layer_columns(*lay*)

Populates the list of columns for a given layer.

set_layers(*elevations*)

Sets mesh layers according to specified vertical layer boundary elevations, from the top down.

set_rectangular_columns(*spacings*)

Sets rectangular mesh columns according to specified lists of horizontal mesh spacings.

set_surface(*surface*)

Sets column layers from surface dictionary (keyed by column indices) or list/array of values for all columns.

setup(*indices=False*)

Sets up internal mesh variables, including node, column and layer indices if needed.

setup_cells()

Sets up cell properties of mesh, layers and columns.

slice_plot(*line='x'*, ***kwargs*)

Creates a 2-D Matplotlib plot of the mesh through a specified vertical slice. The horizontal *line* defining the slice can be either:

- *'x'*: to plot through the mesh centre along the *x*-axis
- *'y'*: to plot through the mesh centre along the *y*-axis
- a number representing an angle (in degrees), clockwise from the *y*-axis, to plot through the centre of the mesh at that angle
- a tuple, list or array of two end-points for the line, each point being itself a tuple, list or array of length 2

Other optional parameters:

- *aspect*: the aspect ratio of the axes (default *'auto'*).
- *axes*: a Matplotlib axes object on which to draw the plot. If not specified, then a new axes object will be created internally.
- *colourmap*: a Matplotlib colourmap object for shading the plot according to the *value* array (default *None*).
- *label*: a string (or *None*, the default) specifying what labels are to be drawn at the centre of each cell. Possible values are *'cell'* (to label cell indices) or *'value'* (to label with the *value* array).
- *label_format*: format string for the labels (default *'%g'*).
- *label_colour*: the colour of the labels (default *'black'*).
- *linecolour*: the colour of the mesh grid (default *'black'*).
- *linewidth*: the line width of the mesh (default *0.2*).
- *value*: a tuple, list or array of values to plot over the mesh, of length equal to the number of cells in the mesh.
- *xlabel*: label string for the plot *x*-axis (default *'x'*).
- *ylabel*: label string for the plot *y*-axis (default *'z'*).

property surface

Array of column surface elevations.

property surface_cells

List of cells at the surface of the mesh.

translate(*shift*)

Translates the mesh by the specified 3-D shift vector (tuple, list or array of length 3).

type_columns(*num_nodes*)

Returns a list of mesh columns of a specified type, i.e. number of nodes.

property volume

Total volume of the mesh.

write(*filename*)

Writes mesh to HDF5 file.

class layermesh.mesh.**node**(*pos*, *index=None*)

Bases: object

2-D mesh node. On creation, the node's horizontal position (and optionally index) are specified.

column

Set containing the column objects the node belongs to.

find(*polygon*, *indices=False*)

Returns self if the node is inside the specified *polygon* (tuple, list or array of 2-D points, each one a tuple, list or array of length 2), otherwise *None*.

index

Integer containing the node's index in the mesh.

pos

Array containing the node's horizontal position.

layermesh.quadtree module

Quadtrees for spatial searching in 2D meshes.

class layermesh.quadtree.**quadtree**(*bounds*, *elements*, *parent=None*)

Bases: object

Quadtree for spatial searching of mesh columns. On creation, the quadtree's bounding box, elements and optional parent quadtree are specified.

Adapted from the quadtree data structure in PyTOUGH.

all_elements

The elements list of the zeroth-generation quadtree.

bounds

The quadtree's bounding box.

child

A list of the quadtree's child quadtrees.

elements

The elements in the quadtree.

generation

The generation index of the quadtree.

leaf(*pos*)

Returns the leaf containing the specified point *pos*.

property num_children

Number of children of the quadtree.

property num_elements

Number of elements in the quadtree.

parent

The quadtree's parent quadtree.

search(*pos*)

Returns the element containing the specified point *pos*.

search_wave(*pos*)

Executes search wave for specified point *pos* on a quadtree leaf.

translate(*shift*)

Translates the quadtree horizontally by the specified shift vector (an array of length 2).

3.1.2 Module contents

PYTHON MODULE INDEX

INDEX

I

L

M

N

O

P