PyMesh Documentation

Release 0.1

Qingnan Zhou

Apr 21, 2017

Contents

PyMesh is a rapid prototyping platform focused on geometry processing. It provides a set of common mesh processing functionalities and interfaces with a number of state-of-the-art open source packages to combine their power seamlessly under a single developing environment.

Mesh process should be simple in python. PyMesh promotes human readable, minimalistic interface and works with native python data structures such as numpy.ndarray.

Load mesh from file:

```
>>> import pymesh
>>> mesh = pymesh.load_mesh("cube.obj");
```

Access mesh vertices:

```
>>> mesh.vertices
array([[-1., -1.,  1.],
       ...
       [ 1.,  1.,  1.]])
>>> type(mesh.vertices)
<type 'numpy.ndarray'>
```

Compute Gaussian curvature for each vertex:

```
>>> mesh.add_attribute("vertex_gaussian_curvature");
>>> mesh.get_attribute("vertex_gaussian_curvature");
array([ 1.57079633,  1.57079633,  1.57079633,  1.57079633,  1.57079633,
        1.57079633,  1.57079633,  1.57079633])
```

CHAPTER 1

Features:

- Read/write 2D and 3D mesh in `.obj`, `.off`, `.ply`, `.stl`, `.mesh` (MEDIT), `.msh` (Gmsh) and `.node/.face/.ele` (Tetgen) formats.
- Support load and save per vertex/face/voxel scalar and vector fields.
- Local mesh processing such edge collapse/split, duplicated vertex/face removal etc.
- Mesh boolean support from CGAL, Cork, Carve, Clipper (2D only) and libigl.
- Mesh generation support from CGAL, Triangle and Tetgen.
- Wire network and inflation of wire networks.
- Finite element matrix assembly. (supports Laplacian, stiffness, mass, etc.)

CHAPTER 2

Contents:

User Guide

Installation

Download the Source:

The source code can be checked out from GitHub:

```
git clone --recursive git@github.com:qnzhou/PyMesh.git
```

The environment variable `PYMESH_PATH` is used by the unit tests to locate the testing data, so be sure to set it up:

```
export PYMESH_PATH=/path/to/PyMesh/
```

Dependencies:

PyMesh is based on the design philosophy that one should not reinvent the wheel. It depends a number of state-of-the-art open source libraries:

- Python: v2.7 or higher
- NumPy: v1.8 or higher
- SciPy: v0.13 or higher
- SWIG: v3.0.5 or higher
- Eigen: v3.2 or higher

The following libraries are not required, but highly recommended. PyMesh provides a thin wrapper to these libraries, and without them certain functionalities would be disabled. Most of these packages can be easily installed using package management softwares for your OS.

- SparseHash: is used to speed up hash grid.

- CGAL: is needed for self-intersection, convex hull, outer hull and boolean computations.
- tetgen: is needed by tetrahedronization and wire inflation.
- libigl: is needed by outer hull, boolean computations and wire inflation.
- cork: is used by boolean computation.
- triangle: is used by triangulation and 2D wire inflation (See triangle_compilation_note).
- qhull: is used for computing convex hull.
- Clipper: is used for 2D boolean operations.
- Carve: is used for 3D boolean operations. Minor modification is added by me for linux/mac compilation.
- GeoGram: is used as a 2D triangle and 3D tetrahedron generation engine.
- Quartet: is used as a 3D tetrahedralization engine.

Environment Variables:

If any dependent libraries are not installed in the default locations, e.g. `/usr/local` and `opt/local`, one needs to set certain environment variables that help PyMesh locate the libraries. PyMesh check the following environment variables:

- `EIGEN_INC`: directory containing the Eigen library.
- `GOOGLEHASH_INCLUDES`: directory containing sparse hash.
- `CGAL_PATH`: path to CGAL library.
- `BOOST_INC`: directory containing boost.
- `LIBIGL_PATH`: path to libigl.
- `CORK_PATH`: path to cork.
- `TETGEN_PATH`: path to tetgen.
- `TRIANGLE_PATH`: path to triangle.
- `QHULL_PATH`: path to qhull.
- `CLIPPER_PATH`: path to clipper.
- `CARVE_PATH`: path to carve.
- `GEOGRAM_PATH`: path to GeoGram.
- `QUARTET_PATH`: path to Quartet.

Building PyMesh:

To compile the optional third party libraries:

```
cd $PYMESH_PATH/third_party
mkdir build
cd build
cmake ..
make
make install
```

Third party dependencies will be installed in `$PYMESH_PATH/python/pymesh/third_party` directory.

It is recommended to build out of source, use the following commands setup building environment:

```
cd $PYMESH_PATH
mkdir build
cd build
cmake ..
```

To only build the C++ libraries without python bindings, change the last command to:

```
cmake -DWITHOUT_SWIG=ON ..
```

PyMesh consists of several modules. The main module defines the core data structures and is used by all other modules. To build the main module and its unit tests:

```
make
make src_tests
```

The other modules are different tools to achieve certain functionalities. Different tools may have different dependencies. A module will be disabled if its dependencies are not met. To build all tools and their unit tests:

```
make tools
make tools_tests
```

Another way is to build each tool separately:

```
# MeshUtils tools
make MeshUtils
make MeshUtils_tests

# EigenUtils tools
make EigenUtils
make EigenUtils_tests

# Assembler tools
make assembler
make assembler_tests

# CGAL tools
make cgal
make cgal_tools

# Boolean tools
make boolean
make boolean_tests

# Convex hull tools
make convex_hull
make convex_hull_tests

# Envolope tools
make envolope
make envolope_tests

# Outer hull tools
make outer_hull
make outer_hull_tests
```

```
# SelfIntersection tools
make self_intersection
make self_intersection_tests

# SparseSolver tools
make SparseSolver
make SparseSolver_tests

# Tetrahedronization tools
make tetrahedronization
make tetrahedronization_tests

# Wire inflation tools
make wires
make wires_tests

# TetGen tools
make tetgen
make tetgen_tests

# Triangle tools
make triangle
make triangle_tests
```

Make sure all unit tests are passed before using the library. Please report unit tests failures on github.

Install PyMesh:

The output of building PyMesh consists a set of C++ libraries and a python module. Installing the C++ library is currently not available. However, installing the python package can be done:

```
./setup.py build # This an alternative way of calling cmake/make
./setup.py install
```

To check PyMesh is installed correctly, one can run the unit tests:

```
python -c "import pymesh; pymesh.test()"
```

Once again, make sure all unit tests are passed, and report any unit test failures.

Basic Usage

PyMesh is rapid prototyping library focused on processing and generating 3D meshes. The *Mesh* class is the core data structure and is used by all modules.

Mesh Data Structure:

In PyMesh, a *Mesh* consists of 3 parts: geometry, connectivity and attributes.

- Geometry consists of vertices, faces and generialized voxels (i.e. a volume element such as tetrahedron or hexahedron). The dimention of the embedding space, face type, voxel type can all be inferred from the geometry data. It is possible for a mesh to consist of 0 vertices or 0 faces or 0 voxels.
- The connectivity contains adjacency information, including vertex-vertex, vertex-face, vertex-voxel, face-face, face-voxel and voxel-voxel adjacencies.

- Attributes are arbirary value field assigned to a mesh. One could assign a scalar or vector for each vertex/face/voxel. There are a number predefined attributes.

Loading Mesh:

From file:

```
>>> mesh = pymesh.load_mesh("model.obj")
```

PyMesh supports parsing the following formats: .obj, .ply, .off, .stl, .mesh, .node, .poly and .msh.

From raw data:

```
>>> # for surface mesh:
>>> mesh = pymesh.form_mesh(vertices, faces)

>>> # for volume mesh:
>>> mesh = pymesh.form_mesh(vertices, faces, voxels)
```

where `vertices`, `faces` and `voxels` are of type `numpy.ndarray`. One vertex/face/voxel per row.

Accessing Mesh Data:

Geometry data can be directly accessed:

```
>>> print(mesh.num_vertices, mesh.num_faces, mesh.num_voxels)
(8, 12, 6)

>>> print(mesh.dim, mesh.vertex_per_face, mesh.vertex_per_voxel)
(3, 3, 4)

>>> mesh.vertices
array([[-1., -1.,  1.],
       ...
       [ 1.,  1.,  1.]])

>>> mesh.faces
array([[0, 1, 2],
       ...
       [4, 5, 6]])

>>> mesh.voxels
array([[0, 1, 2, 3],
       ...
       [4, 5, 6, 7]])
```

Connectivity data is disabled by default because it is often not needed. To enable it:

```
>>> mesh.enable_connectivity();
```

The connectivity information can be queried using the following methods:

```
>>> mesh.get_vertex_adjacent_vertices(vi);
>>> mesh.get_vertex_adjacent_faces(vi);
>>> mesh.get_vertex_adjacent_voxels(vi);
```

```
>>> mesh.get_face_adjacent_faces(fi);
>>> mesh.get_face_adjacent_voxels(fi);

>>> mesh.get_voxel_adjacent_faces(Vi);
>>> mesh.get_voxel_adjacent_voxels(Vi);
```

Using Attributes:

Attributes allow one to attach a scalar or vector fields to the mesh. For example, vertex normal could be stored as a mesh attribute where a normal vector is associated with each vertex. In addition to vertices, attribute could be associated with face and voxels. To create an attribute:

```
>>> mesh.add_attribute("attribute_name");
```

This creates an empty attribute (of length 0) called `attribute_name`. To assign value to the attribute:

```
>>> val = np.ones(mesh.num_vertices);
>>> mesh.set_attribute("attribute_name", val);
```

Notice that the `val` variable is a native python `numpy.ndarray`. The length of the attribute is used to determine whether it is a scalar field or vector field. The length is also used to determine whether the attribute is assigned to vertices, faces or voxels.

To access a defined attribute:

```
>>> attr_val = mesh.get_attribute("attribute_name");
>>> attr_val
array([ 1.0,  1.0,  1.0, ...,  1.0, 1.0,  1.0])
```

The following vertex attributes are predifined:

- `vertex_normal`: A vector field representing surface normals. Zero vectors are assigned to vertices in the interior.
- `vertex_volume`: A scalar field representing the lumped volume of each vertex (e.g. 1/4 of the total volume of all neighboring tets for tetrahedron mesh.).
- `vertex_area`: A scalar field representing the lumped surface area of each vertex (e.g. 1/3 of the total face area of its 1-ring neighborhood).
- `vertex_laplacian`: A vector field representing the discretized Laplacian vector.
- `vertex_mean_curvature`: A scalar field representing the mean curvature field of the mesh.
- `vertex_gaussian_curvature`: A scalar field representing the Gaussian curvature field of the mesh.
- `vertex_index`: A scalar field representing the index of each vertex.
- `vertex_valance`: A scalar field representing the valance of each vertex.
- `vertex_dihedral_angle`: A scalar field representing the max dihedral angle of all edges adjacent to this vertex.

The following face attributes are predefined:

- `face_area`: A scalar field representing face areas.
- `face_centroid`: A vector field representing the face centroids (i.e. average of all corners).
- `face_circumcenter`: A vector field representing the face circumcenters (defined for triangle faces only).

- `face_index`: A scalar field representing the index of each face.
- `face_normal`: A vector field representing the normal vector of each face.
- `face_voronoi_area`: A vector field representing the voronoi area of each corner of the face.

The following voxel attributes are predefined:

- `voxel_index`: A scalar field representing the index of each voxel.
- `voxel_volume`: A scalar field representing the volume of each voxel.
- `voxel_centroid`: A scalar field representing the centroid of each voxel (i.e. average of all corners of a voxel).

Predefined attribute does not need to be set:

```
>>> mesh.add_attribute("vertex_area")
>>> mesh.get_attribute("vertex_area")
array([ 0.56089278,  0.5608997 ,  0.57080866, ...,  5.62381961,
        2.12105028,  0.37581711])
```

Notice that attribute values are always stored as a 1D array. For attributes that represent vector/tensor fiels, the attribute values are the flattened version of the vector field:

```
>>> mesh.add_attribute("vertex_normal")
>>> mesh.get_attribute("vertex_normal")
array([ 0.35735435, -0.49611438, -0.79130802, ..., -0.79797784,
        0.55299134, -0.23964964])
```

If an attribute is known to be a per-vertex attribute, one can:

```
>>> mesh.get_vertex_attribute("vertex_normal")
array([[ 0.35735435, -0.49611438, -0.79130802],
       [ 0.41926554, -0.90767626, -0.01844495],
       [-0.64142577,  0.76638469, -0.03503568],
       ...,
       [-0.64897662, -0.64536558, -0.40290522],
       [-0.92207726, -0.10573231, -0.37228242],
       [-0.79797784,  0.55299134, -0.23964964]])
```

where attribute values are returned as a 2D matrix. Each row represents the value per vertex.

Similarly, per-face and per-voxel attribute can be retrieved using `get_face_attribute()` and `get_voxel_attribute()` methods.

To retrieve the names of all defined attributes for a given mesh:

```
>>> mesh.get_attribute_names()
("attribute_name", "vertex_area", "vertex_normal")
```

Saving Mesh:

The following formats are supported for saving meshes: .obj, .off, .ply, .mesh, .node, .poly, .stl and .msh. However, saving in .stl format is strongly discouraged because STL files use more disk space and stores less information. To save a mesh:

```
>>> pymesh.save_mesh("filename.obj", mesh);
```

For certain formats (e.g. .ply, .msh, .stl), it is possible to save either as an ASCII file or a binary file. By default, PyMesh will always use the binary format. To save in ASCII, just set the `ascii` argument:

```
>>> pymesh.save_mesh("filename.obj", mesh, ascii=True)
```

In addition, vertex position can be saved using `double` or `float`. By default, PyMesh saves in `double`, to save using `float`:

```
>>> pymesh.save_mesh("filename.obj", mesh, use_float=True)
```

Mesh attributes can also be saved in .msh and .ply formats. To save with attributes:

```
>>> pymesh.save_mesh("filename.msh", mesh, attribute_name_1, attribute_name_2, ...)
```

To save with all defined attributes:

```
>>> pymesh.save_mesh("filename.msh", mesh, *mesh.get_attribute_names())
```

It is also possible to save from raw vertices, faces and voxels:

```
>>> # For surface mesh
>>> pymesh.save_mesh_raw("filename.ply", vertices, faces)

>>> # For volume mesh
>>> pymesh.save_mesh_raw("filename.ply", vertices, faces, voxels)

>>> # In ascii and using float
>>> pymesh.save_mesh_raw("filename.ply", vertices, faces, voxels,\
        ascii=True, use_float=True)
```

Mesh Processing

It is often necessary to change the mesh. PyMesh has built-in capabilities of commonly used operations.

Collapse Short Edges:

To collapse all edges shorter than or equal to `tol`:

```
>>> mesh, info = pymesh.collapse_short_edges(mesh, tol)
```

The function returns two things: a new mesh with all short edges removed, and some extra information:

```
>>> info.keys()
['num_edge_collapsed']

>>> info["num_edge_collapse"]
124

>>> mesh.attribute_names
['face_sources']

>>> mesh.get_attribute("face_sources")
array([    0,     1,     2, ..., 20109, 20110, 20111])
```

The `face_sources` attribute maps each output face to the index of the source face from the input mesh.

One can even perform this function on a raw mesh:

```
>>> vertices, faces, info = pymesh.collapse_short_edges_raw(
...     vertices, faces, tol)
```

In addition to setting an absolute threshold, one can use a relative threshold based on the average edge length:

```
>>> mesh, __ = pymesh.collapse_short_edges(mesh, rel_threshold=0.1)
```

In the above example, all edges shorter than or equal to 10% of the average edge length are collapsed.

It is well known that sharp features could be lost due to edge collapse. To avoid destroying sharp features, turn on the `preserve_feature` flag:

```
>>> mesh, __ = pymesh.collapse_short_edges(mesh, tol,
...     preserve_feature=True)
```

One of the main applications of this method is to simplify overly triangulated meshes. As shown in the following figure, the input mesh (top) is of very high resolution near curvy regions. With `pymesh.collapse_short_edges`, we can create a coarse mesh (bottom) to approximate the input shape. The quality of the approximation depends heavily on the value of `tol`.

Split Long Edges:

Another very useful but rarely implemented mesh processing operation is to split long edges. To split all edges longer than `tol`:

```
>>> mesh, info = pymesh.split_long_edges(mesh, tol)
```

The return values consist of the new mesh and a dummy information field for future usage:

```
>>> info.keys()
{}
```

The returned mesh contains all the vertices from input mesh and newly inserted vertices. Certain faces may be split. Unlike standard subdivision algorithm, the algorithm only split faces that contain long edges and leaves the rest alone.

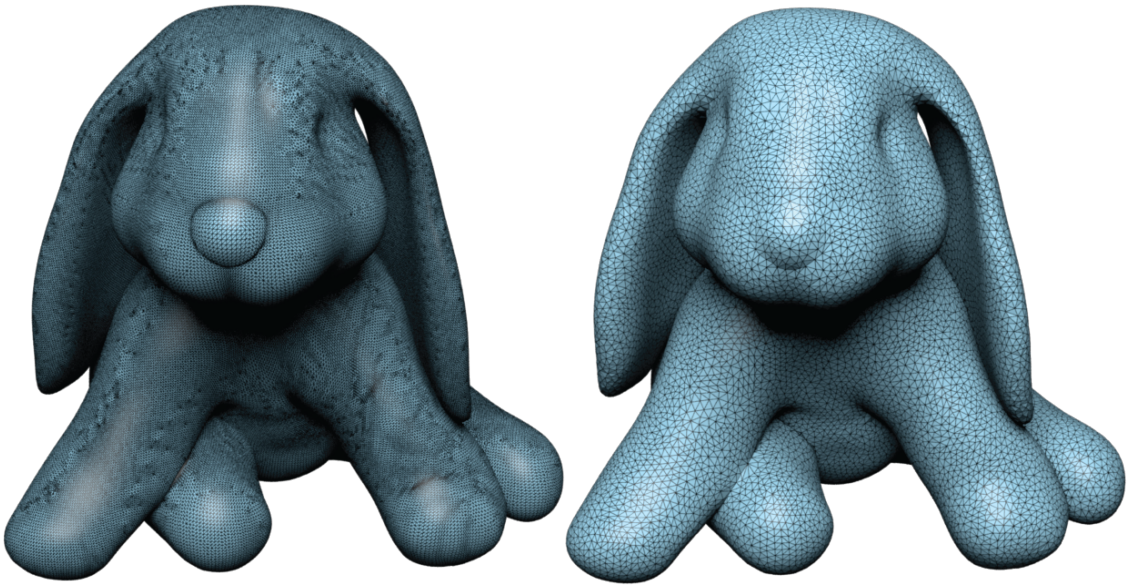
It is also possible to operate on a raw mesh:

```
>>> vertices, faces, info = pymesh.split_long_edges(mesh, tol)
```

This method is often used to elimiate long edges appearing in sliver triangles. The following figure shows its effect.

Remeshing:

It is possible to completely remesh the input shape by calling `pymesh.collapse_short_edges` and `pymesh.split_long_edges` iteratively in an alternating fashion. The script fix_mesh.py is based on this idea. Its effects can be seen in a remesh of the Ducky The Lop Eared Bunny example:



Remove Isolated Vertices:

To remove vertices that is not part of any face or voxel:

```
>>> mesh, info = pymesh.remove_isolated_vertices(mesh)
```

In addition to the output mesh, a information dictionary is returned:

```
>>> info.keys()
['ori_vertex_index', 'num_vertex_removed']

>>> info["ori_vertex_index"]
array([     0,      1,      2, ..., 167015, 167016, 167017])

>>> info["num_vertex_removed"]
12
```

As usual, there is a version that operates directly on the raw mesh:

```
>>> vertices, face, __ = pymesh.remove_isolated_vertices_raw(
...     vertices, faces)
```

Remove Duplicated Vertices:

Sometimes, one may need to merge vertices that are coinciding or close-by measured in Euclidean distance. For example, one may need to zip the triangles together from a triangle soup. To achieve it:

```
>>> mesh, info = pymesh.remove_duplicated_vertices(mesh, tol)
```

The argument `tol` defines the tolerance within which vertices are considered as duplicates. In addition to the output mesh, some information is also returned:

```
>>> info.keys()
['num_vertex_merged', 'index_map']

>>> info["num_vertex_merged"]
5

>>> info["index_map"]
array([     0,      1,      2, ..., 153568, 153569, 153570])
```

By default, all duplicated vertices are replaced by the vertex with the smallest index. It is sometimes useful to specify some vertices to be more important than other and their coordinates should be used as the merged vertices in the output. To achieve this:

```
>>> weights = mesh.vertices[:, 0];
>>> mesh, info = pymesh.remove_duplicated_vertices(mesh, tol,
...     importance=weights)
```

In the above example, we use the X coordinates as the importance weight. When closeby vertices are merged, the coordinates of the vertex with the highest X values are used.

As usual, one can operate directly on raw meshes:

```
>>> vertices, faces, info = pymesh.remove_duplicated_vertices_raw(
...     vertices, faces, tol)
```

Remove Duplicated Faces:

It is also useful to remove duplicated faces:

```
>>> mesh, info = pymesh.remove_duplicated_faces(mesh)
```

The resulting mesh and some information is returned:

```
>>> info.keys()
['ori_face_index']

>>> info["ori_face_index"]
array([    0,     1,     2, ..., 54891, 54892, 54893])
```

The field `ori_face_index` provides the source vertex index for each output vertex.

To operate on raw meshes:

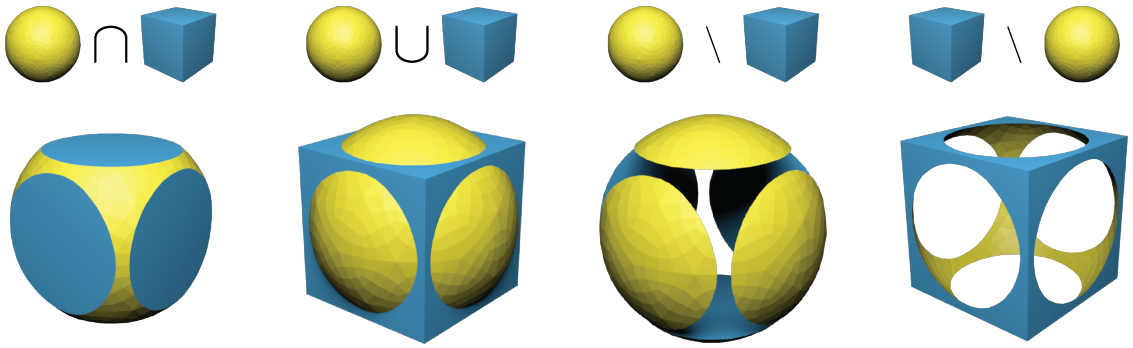
```
>>> vertices, faces, info = pymesh.remove_duplicated_faces(
...     vertices, faces)
```

Mesh Boolean

Boolean operation is one of the fundamental operations for 3D modeling. It combines two or more solid shapes (say A and B) by checking if a point x lies inside of each solid. Four commonly used binary boolean operations are:

- Union: $A \cup B := \{x \in \mathbb{R}^3 \mid x \in A \text{ and } x \in B\}$
- Intersection: $A \cap B := \{x \in \mathbb{R}^3 \mid x \in A \text{ or } x \in B\}$
- Difference: $A \setminus B := \{x \in \mathbb{R}^3 \mid x \in A \text{ and } x \notin B\}$
- Symmetric difference: $A \text{ XOR } B := (A \setminus B) \cup (B \setminus A)$

The following figure illustrates the output of boolean operations on a sphere and a cube:



Boolean Interface:

PyMesh provides support for all four operations through third party boolean *engines*. For example, computing the union of `mesh_A` and `mesh_B` can be achieved with the following snippet:

```
>>> mesh_A = pymesh.load_mesh("A.obj")
>>> mesh_B = pymesh.load_mesh("B.obj")
>>> output_mesh = pymesh.boolean(mesh_A, mesh_B,
```

```
...                                 operation="union",
...                                 engine="igl")
```

The interface is very minimal and self-explanatory. The available operations are `"union"`, `"intersection"`, `"difference"` and `"symmetric_difference"`. PyMesh supports the following boolean engines:

- `"igl"`: Boolean module from libigl, which is also the default engine for 3D inputs.
- `"cgal"`: Naf polyhedron implementation from CGAL.
- `"cork"`: Cork boolean library.
- `"carve"`: Carve boolean library.
- `"corefinement"`: The unpublished boolean engine also from CGAL.
- `"clipper"`: 2D boolean engine for polygons, the default engine for 2D inputs.

The following attributes are defined in the `output_mesh`:

- `source`: A per-face scalar attribute indicating which input mesh an output face belongs to.
- `source_face`: A per-face scalar attribute representing the combined input face index of an output face, where combined input faces are simply the concatenation of faces from `mesh_A` and `mesh_B`.

A Simple Example:

As a simple example, we are going to operate on the following objects:

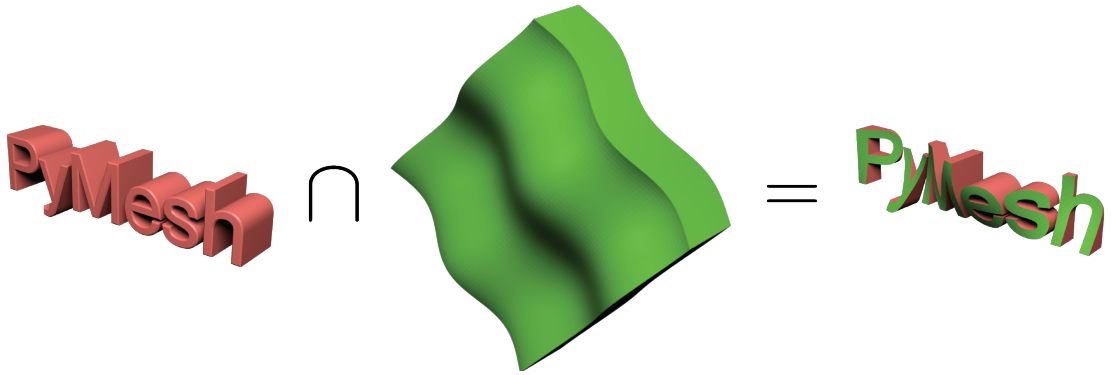
- Mesh A (pymesh.ply) contains the extruded text `PyMesh`.
- Mesh B (plate.ply) contains an extruded wavy plate.

To compute their intersection:

```
>>> A = pymesh.load_mesh("pymesh.ply")
>>> B = pymesh.load_mesh("plate.ply")
>>> intersection = pymesh.boolean(A, B, "intersection")

>>> # Checking the source attribute
>>> intersection.attribute_names
('source', 'source_face')
>>> intersection.get_attribute("source")
array([ 1.,  1.,  0., ...,  1.,  1.,  1.])
```

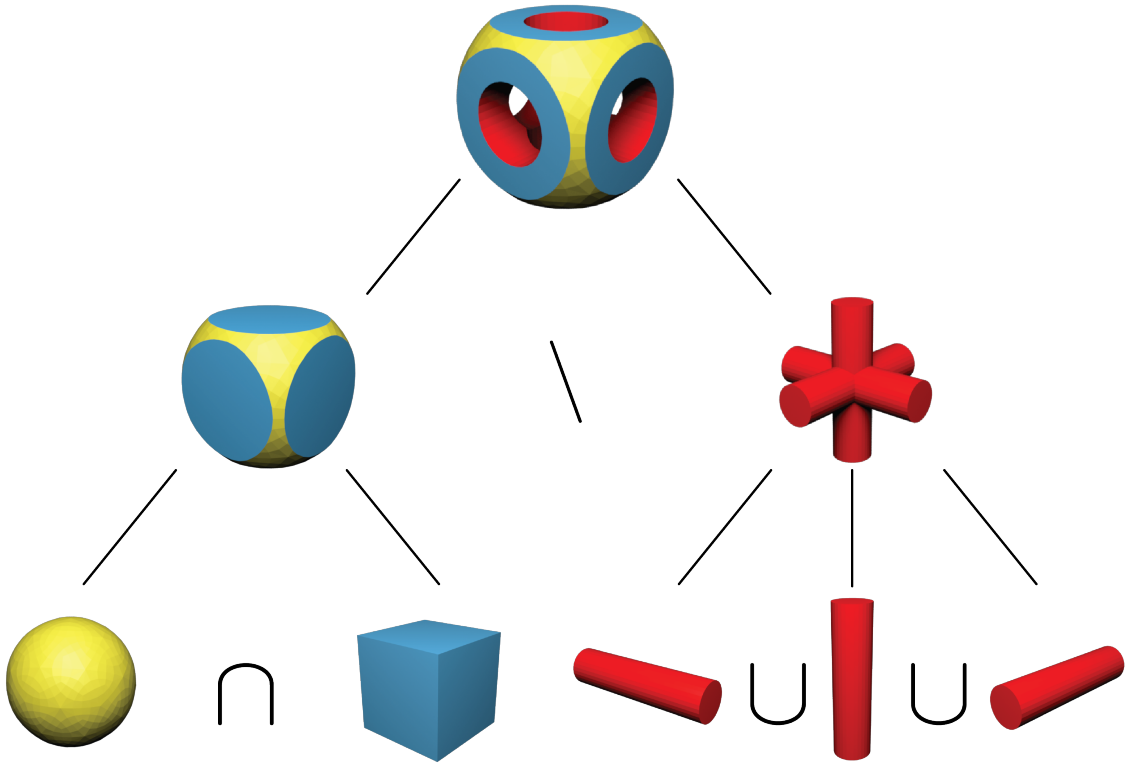
The operation is illustrated in the following figure:



The attribute `source` tracks the *source* of each output face. 0 means the output face comes from the first operand, i.e. `pymesh.ply`, and 1 means it is from the second operand, i.e. `plate.ply`. The `source` attribute is useful for assigning the corresponding colors in the output mesh.

CSG Tree:

While binary boolean operations are useful, it is often necessary to perform a number of operations in order to create more complex results. A Constructive Solid Geometry tree, aka.CSG tree, is designed for this purpose.



As illustrated in the figure above, CSG tree provides a structured way of building complex shapes from simple ones. Each node in the tree represents a 3D shape. Leaf nodes represent user input shapes. A non-leaf node consists of a boolean operation and a number of child nodes. The shape it represents can be obtained by performing the specified boolean operation on shapes from its children. In particular, `union` and `intersection` node can have any number of children (i.e. N-ary union and N-ary intersection), but `difference` and `symmetric_difference` nodes must have exactly two children.

PyMesh represents CSG tree using `pymesh.CSGTree` class. Given the input meshes, one can construct and evaluate a CST tree using the following code:

```
>>> ball = pymesh.load_mesh("ball.stl")
>>> box = pymesh.load_mesh("box.stl")
>>> x = pymesh.load_mesh("x.stl")
>>> y = pymesh.load_mesh("y.stl")
>>> z = pymesh.load_mesh("z.stl")

>>> csg = pymesh.CSGTree({
        "difference": [
            { "intersection": [{"mesh": box}, {"mesh": ball}] },
            { "union": [{"mesh": x}, {"mesh": y}, {"mesh": z}]}
        ]
    })
```

```
>>> output = csg.mesh
```

Notice that the constructor of `CSGTree` takes a python dictionary as argument. The entire tree structure is captured in the dictionary. The context free grammar for this dictionary is:

```
Node -> {Operation : Children}
Node -> {"mesh": Mesh}
Node -> CSGTree
Children -> [Node, Node, ...]
Operation -> "union" | "intersection" | "difference" | "symmetric_difference"
```

where `Mesh` is a `pymesh.Mesh` object and `CSGTree` is a `pymesh.CSGTree` object. One can construct the entire tree all together as shown above or build up the tree incrementally:

```
>>> left_tree = pymesh.CSGTree({
        "intersection": [{"mesh": box}, {"mesh": ball}]
    })
>>> right_tree = pymesh.CSGTree({
        "union": [{"mesh": x}, {"mesh": y}, {"mesh": z}]
    })
>>> csg = pymesh.CSGTree({
        "difference": [left_tree, right_tree]
    })

>>> left_mesh = left_tree.mesh
>>> right_mesh = right_tree.mesh
>>> output = csg.mesh
```

Wire Inflation

Overview:

The goal of `wires` package is to provide an easy way of modeling frame structures. A frame structure can be uniquely define by 3 parts:

- Vertex positions
- Topology/Connectivity
- Edge/vertex thickness

Given these 3 parts as input, we implement a very efficient algorithm proposed by George Hart to generate the output triangular mesh.

The `wires` involves just 3 main classes: *WireNetwork*, *Inflator* and *Tiler*. Understanding these 3 classes would allow one to generate wide variety of frame structures. In a nutshell, vertex positions and topology are encoded in the *WireNetwork* data structure. The *Inflator* class takes a *WireNetwork* object and its corresponding thickness assignment as input, and it outputs an triangular mesh. The *Tiler* class takes a *WireNetwork* object as a unit pattern and tile it according to certain rules, and its output is the tiled *WireNetwork* object.

WireNetwork:

`WireNetwork` class represents the vertex positions and topology of a frame structure. It can be easily modeled by hand or using tools such as blender.

Construction from data:

To create a *WireNetwork* object, we just need to provide a set of vertices and a set of edges:

```
>>> vertices = np.array([
...     [0, 0, 0],
...     [1, 0, 0],
...     [0, 1, 0],
...     [1, 1, 0]
...     ])
>>> edges = np.array([
...     [0, 1],
...     [1, 3],
...     [2, 3],
...     [2, 0]
...     ]);
>>> wire_network = pymesh.wires.WireNetwork.create_from_data(
...     vertices, edges)
```

Notice that `edges` is a list of vertex index pairs, and vertex index starts from 0.

Construction from file:

Alternatively, one can use `.obj` format to encode a wire networks:

```
# Filename: test.wire
v 0.0 0.0 0.0
v 1.0 0.0 0.0
v 0.0 1.0 0.0
v 1.0 1.0 0.0
l 1 2
l 2 4
l 3 4
l 3 1
```

Lines starting with `v` are specifies vertex coordinates, and lines starting with `l` are edges. Notice that indices starts from 1 in `.obj` files. One advantage of using the `.obj` format to store wire network is that it can be opened directly by Blender. However, to distinguish with triangular mesh, I normally change the suffix to `.wire`.

To create a wire network from file:

```
>>> wire_network = pymesh.wires.WireNetwork.create_from_file(
...     "test.wire")
```

Empty wire network and update data:

Sometimes it is useful to create an empty wire network (0 vertices, 0 edges):

```
>>> empty_wires = pymesh.wires.WireNetwork.create_empty()
```

Once created, the vertices and edges of a *WireNetwork* are generally read-only because updating the geometry typically invalidates vertex and edge attributes such as edge lengths. However, it is possible to assign an entirely new set of vertices and edges to a *WireNetwork* object using the `load` and `load_from_file` method:

```
>>> wire_network.load(vertices, edges)

>>> wire_network.load_from_file("test.wire")
```

Save to file:

To save a wire network to file:

```
>>> wire_network.write_to_file("debug.wire")
```

Accessing vertices and edges:

Once a `WireNetwork` object is created, one can access the vertices and edges directly:

```
>>> wire_network.dim
3
>>> wire_network.num_vertices
4
>>> wire_network.vertices
array([[ 0.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 1.,  1.,  0.]])

>>> wire_network.num_edges
4
>>> wire_network.edges
array([[0, 1],
       [1, 3],
       [2, 3],
       [2, 0]])
```

Vertex adjacency:

One can easily access vertex adjacency information by `get_vertex_neighbors` method:

```
>>> wire_network.get_vertex_neighbors(vertex_index)
array([1, 9])
```

This method can also be used for computing vertex valance (i.e. the number of neighboring vertices).

Attributes:

Just like the *Mesh* class, it is possible to define attribute to represent scalar and vector fields associated with each vertex and edge. For example:

```
>>> vertex_colors = np.zeros((wire_network.num_vertices, 3));
>>> wire_network.add_attribute("vertex_color", vertex_colors)
>>> print(wire_network.get_attribute("vertex_color"));
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
```

```
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

All attribute names can be retrieved using the `attribute_names` attribute:

```
>>> wire_network.attribute_name
("vertex_color")
```

Symmetry orbits:

It is sometimes important to compute the symmetry orbits of the wire network:

```
>>> wire_network.compute_symmetry_orbits()
```

This command adds 4 attributes to the wire network:

- `vertex_symmetry_orbit`: Per-vertex scalar field specifying the orbit each vertex belongs to . Vertices from the same orbit can be mapped to each other by reflection with respect to axis-aligned planes.
- `vertex_cubic_symmetry_orbit`: Per-vertex scalar field specifying the cubic orbit each vertex belongs to. Vertices from the same cubic orbit can be mapped to each other by all reflection symmetry planes of a unit cube.
- `edge_symmetry_orbit`: Per-edge scalar field specifying the orbit each edge belongs to. Edges from the same orbit can be mapped to each other by reflection with respect to axis-aligned planes.
- `edge_cubic_symmetry_orbit`: Per-edge scalar field specifying the cubic orbit each edge belongs to. Edges from the same cubic orbit can be mapped to each other by reflection with respect to reflection symmetry planes of a unit cube.

These attributes can be access via `get_attribute()` method:

```
>>> wire_network.get_attribute("vertex_symmetry_orbit")
array([ 0.,  0.,  0.,  0., 1., 1., 1., 1.])
```

In the example above, vertex 0 to 3 belongs to orbit 0, and vertex 4 to 7 belongs to orbit 1.

Miscellaneous functions:

The `WireNetwork` class also have a list of handy built-in functionalities.

To access axis-aligned bounding box:

```
>>> bbox_min, bbox_max = wire_network.bbox
>>> bbox_min
array([ 0.  ,  0.  , 0.])
>>> bbox_max
array([ 1.  ,  1.  , 0.])
>>> wire_network.bbox_center
array([0.5, 0.5, 0.0])
```

To access the centroid of the wire network (average of the vertex locations):

```
>>> wire_network.centroid
array([0.5, 0.5, 0.0])
```

To access the edge lengths:

```
>>> wire_network.edge_lengths
array([1.0, 1.0, 1.0, 1.0])
>>> wire_network.total_wire_length
4.0
```

To recursively trim all dangling edges (edges with at least one valance 1 end points):

```
>>> wire_network.trim()
```

To offset each vertex:

```
>>> offset_vectors = np.random.rand(
...     wire_network.num_vertices, wire_network.dim)
>>> wire_network.offset(offset_vectors);
```

To center the wire network at the origin (such that its bounding box center is the origin):

```
>>> wire_network.center_at_origin()
```

Wire Inflation:

Uniform thickness:

Wire inflation refers to the process of converting a *WireNetwork* plus some thickness assignment to a triangular mesh. The inflation logic is encapsulated in the *Inflator* class:

```
>>> inflator = pymesh.wires.Inflator(wire_network)
```

Thickness is just a scalar field. It can be assigned to each vertex or to each edge. Here are some example to assign uniform thickness to vertices and edges:

```
>>> # Assign each vertex with thickness 0.5mm
>>> inflator.inflate(0.5, per_vertex_thickness=True)
>>> mesh = inflator.mesh

>>> # Assign each edge with thickness 0.5mm
>>> inflator.inflate(0.5, per_vertex_thickness=False)
>>> mesh = inflator.mesh
```

The output mesh look the same due to uniform thickness.

Because per-vertex and per-edge uniform thickness assignments produce the same output, one does not need to explicity specify the `per_vertex_thickness` flag:

```
>>> inflator.inflate(0.5)
>>> mesh = inflator.mesh
```

Variable thickness:

It is also possible to assign a thickness value per-vertex or per-edge:

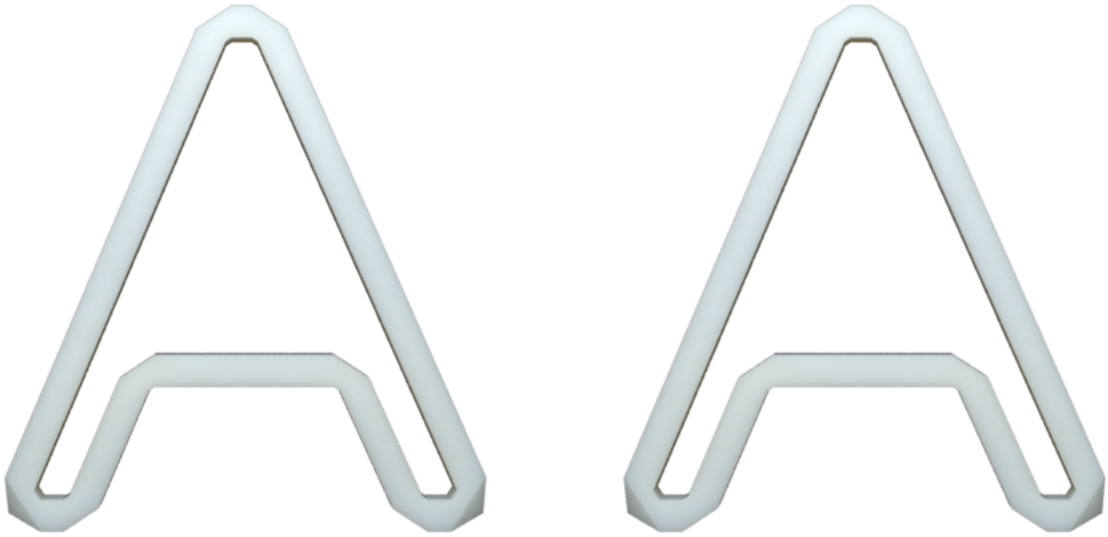


Fig. 2.1: Inflation output for uniform vertex thickness (left) and uniform edge thickness (right).

```
>>> # Assign each vertex with thickness 0.1 to 0.6
>>> thickness = np.arange(wire_network.num_vertices) / \
...     wire_network.num_vertices * 0.5 + 0.1
>>> inflator.inflate(thickness, per_vertex_thickness=True)
>>> mesh = inflator.mesh

>>> # Assign each edge with thickness 0.1 to 0.6
>>> thickness = np.arange(wire_network.num_edges) / \
...     wire_network.num_edges * 0.5 + 0.1
>>> inflator.inflate(thickness, per_vertex_thickness=False)
>>> mesh = inflator.mesh
```

and the output meshes looks like the following:

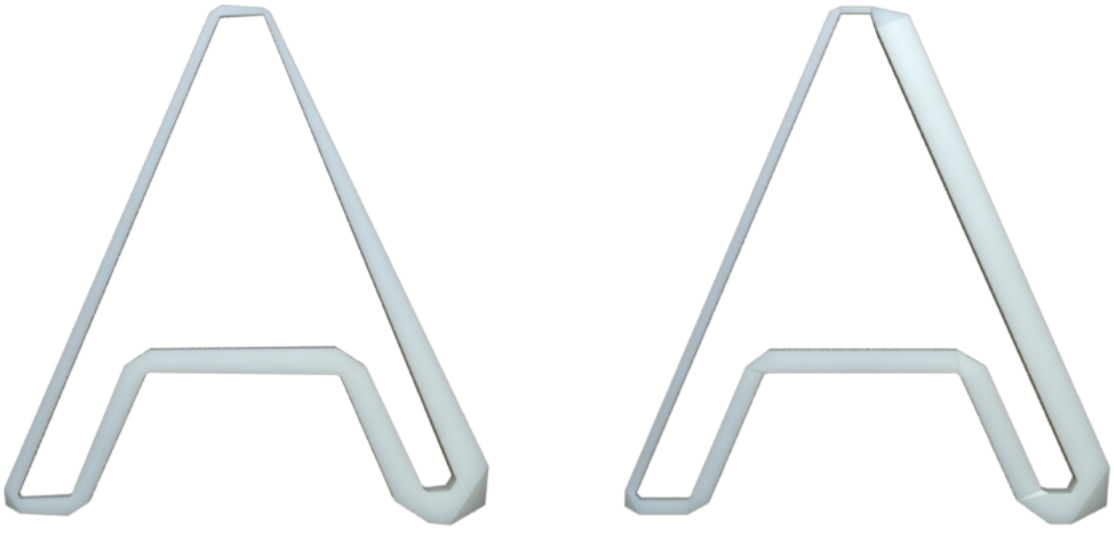


Fig. 2.2: Inflation output for per-vertex (left) and per-edge (right) thickness.

Refinement:

As one may notice from the figure above, the inflated mesh could contain sharp corners. This may be undesirable sometimes. Fortunately, *Inflator* class has refinement built-in:

```
>>> thickness = np.arange(wire_network.num_vertices) / \
...     wire_network.num_vertices * 0.5 + 0.1
>>> inflator.set_refinement(2, "loop")
>>> inflator.inflate(thickness, per_vertex_thickness=True)
>>> mesh = inflator.mesh
```

The above example refines the output mesh by applying `loop` subdivision twice. This create a smooth inflated mesh:

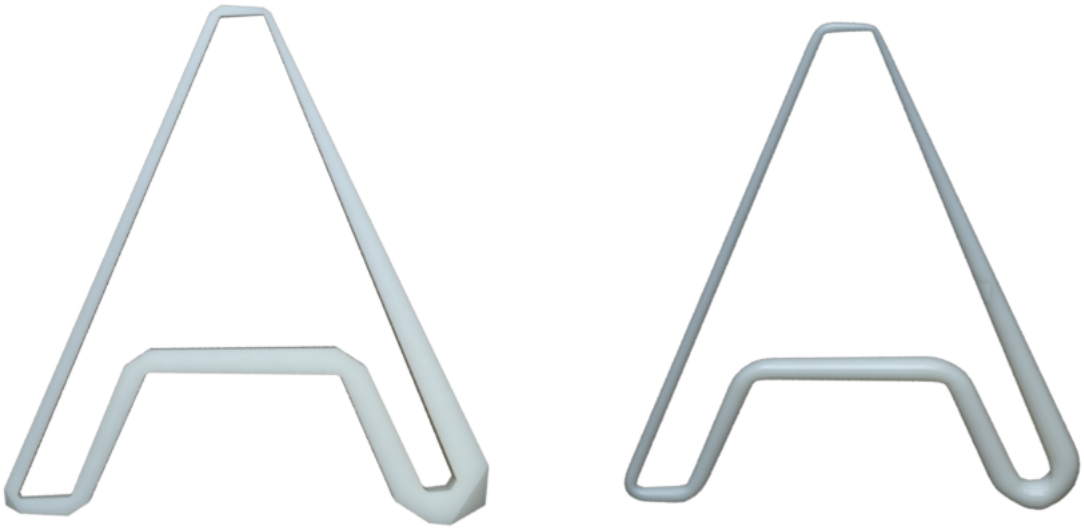


Fig. 2.3: Inflation output without (left) and with (right) `loop` refinement.

Another refinement method is the `simple` refinement. The `simple` refinement does not smooth the geometry but adds more triangles.

Wire profile:

By default, each wire is inflated to a rectangular pipe with square cross sections. It is possible to use any regular N-gon as the cross section by setting the wire profile:

```
>>> # Hexagon
>>> inflator.set_profile(6)
>>> mesh = inflator.mesh

>>> # Triangle
>>> inflator.set_profile(3)
>>> mesh = inflator.mesh
```

Tiling:

The *Inflator* class is capable of inflating arbitrary wire networks. One particular important use case is to inflate a tiled network. The *Tiler* class takes a single *WireNetwork* object as input and generate a tiled wire network that can be later

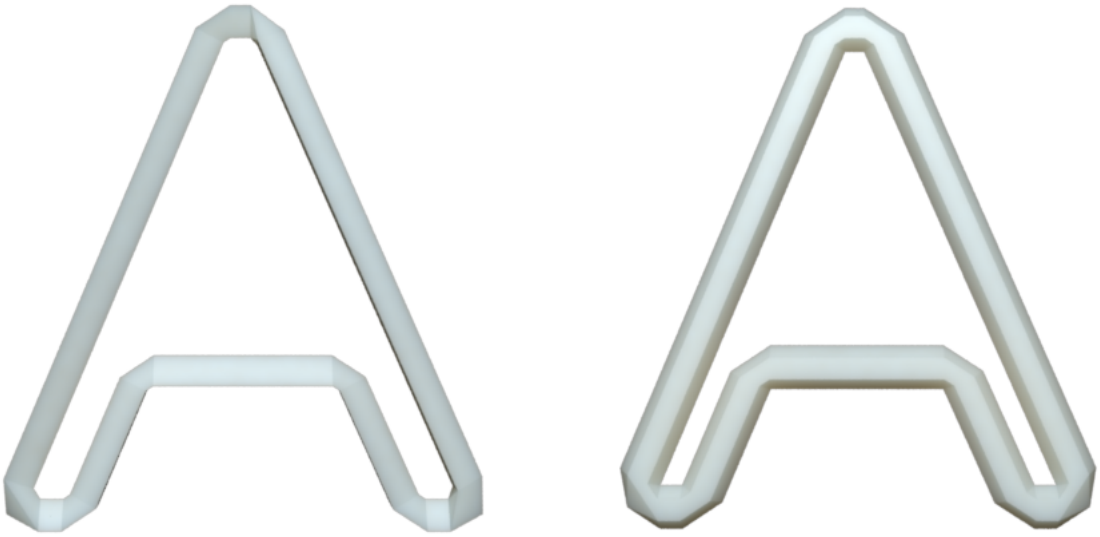


Fig. 2.4: Inflation with triangle profile (left) and hexagon profile (right).

inflated. There are several ways to perform tiling.

Regular tiling:

Regular tiling refers to tiling according to a regular grid. To tile a cube of size 15 with a 3x3x3 tiling of a given wire network (e.g. similar to putting a wire network in each cell of a rubic cube):

```
>>> tiler = Tiler(wire_network)
>>> box_min = np.zeros(3)
>>> box_max = np.ones(3) * 15.0
>>> reps = [3, 3, 3]
>>> tiler.tile_with_guide_bbox(box_min, box_max, reps)
>>> tiled_wires = tiler.wire_network
```

The output `tiled_wires` (inflated with thickness 0.5 and refined twice) looks like the following:

Mesh guided tiling:

It is also possible to tile according to any hexahedron mesh. For example, provided an L-shaped hex mesh:

```
>>> guide_mesh = pymesh.load_Mesh("L_hex.msh")
>>> tiler = Tiler(wire_network)
>>> tiler.tile_with_guide_mesh(guide_mesh)
>>> tiled_wires = tiler.wire_network
```

The output (inflated with thickness 0.5 and refined twice) looks like:

In fact, the guide hex mesh does not need to be axis-aligned. The single cell wire network would be warped to fit inside each hex using tri/bi-linear interpolation. An example is coming soon.

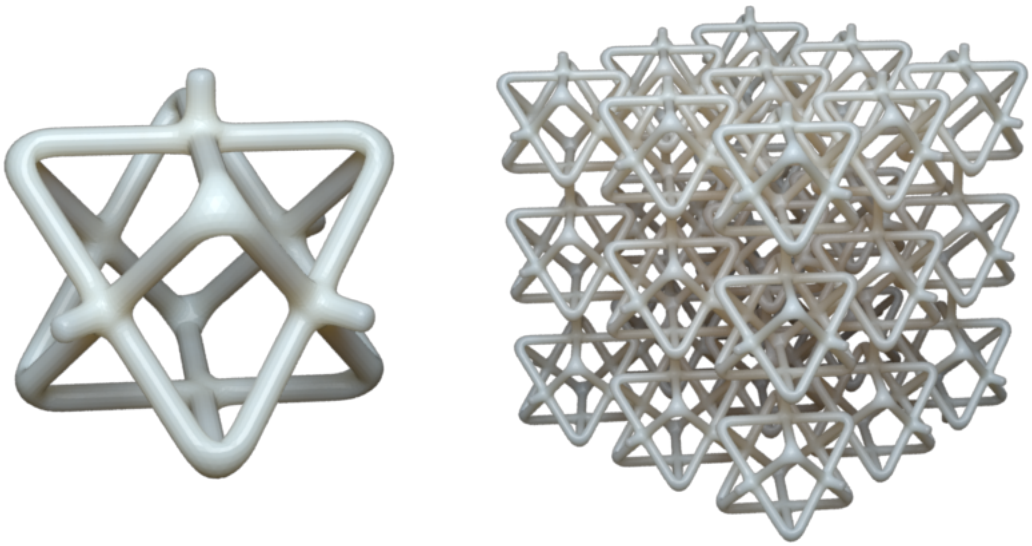


Fig. 2.5: A single cell wire network (left) and the corresponding 3x3x3 tiling (right).

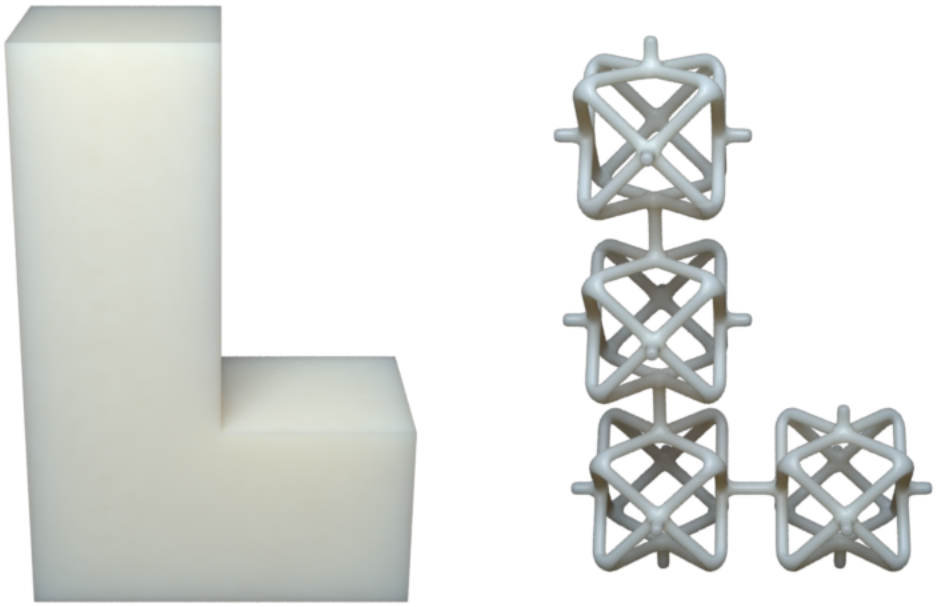


Fig. 2.6: Guide hex mesh (left) and the tiled result (right).

PyMesh API Reference

Mesh Data Structure

class pymesh.**Mesh**(*raw_mesh*)

A generic representation of a surface or volume mesh.

vertices
A $(N_v \times D)$ array of vertex coordinates, where N_v is the number of vertices, D is the dimention of the embedding space (D must be either 2 or 3).

faces
A $(N_f \times F_d)$ array of vertex indices that represents a generalized array of faces, where N_f is the number of faces, F_d is the number of vertex per face. Only triangles ($F_d = 3$) and quad ($F_d = 4$) faces are supported.

voxels
A $(N_V \times V_d)$ array of vertex indices that represents an array of generalized voxels, where N_V is the number of voxels, V_d is the number of vertex per voxel. Only tetrahedron ($V_d = 4$) and hexahedron ($V_d = 8$) are supported for now.

num_vertices
Number of vertices (N_v).

num_faces
Number of faces (N_f).

num_voxels
Number of voxels (N_V).

dim
Dimention of the embedding space (D).

vertex_per_face
Number of vertices in each face (F_d).

vertex_per_voxel
Number of vertices in each voxel (V_d).

attribute_names
Names of all attribute associated with this mesh.

bbox
A $(2 \times D)$ array where the first row is the minimum values of each vertex coordinates, and the second row is the maximum values.

nodes
Same as *vertices*.

elements
Array of elements of the mesh. Same as *faces* for surface mesh, or *voxels* for volume mesh.

num_nodes
Number of nodes.

num_elements
Number of elements.

nodes_per_element
Number of nodes in each element.

element_volumes
An array representing volumes of each element.

num_components
Number of vertex-connected_components.

num_surface_components
Number of edge-connected components.

num_volume_components
Number of face-connected components.

vertices_ref
Same as *vertices* but the return value is a reference.

faces_ref
Same as *faces* but the return value is a reference.

voxels_ref
Same as *voxels* but the return value is a reference.

nodes_ref
Same as *nodes* but the return value is a reference.

elements_ref
Same as *elements* but the return value is a reference.

add_attribute(*name*)
Add an attribute to mesh.

get_attribute(*name*)
Return attribute values in a flattened array.

get_attribute_names()
Get names of all attributes associated with this mesh.

get_attribute_ref(*name*)
Return attribute values in a flattened array.

get_face_attribute(*name*)
Same as *get_attribute()* but reshaped to have *num_faces* rows.

get_vertex_attribute(*name*)
Same as *get_attribute()* but reshaped to have *num_vertices* rows.

get_voxel_attribute(*name*)
Same as *get_attribute()* but reshaped to have *num_voxels* rows.

has_attribute(*name*)
Check if an attribute exists.

is_closed()
Return true iff this mesh is closed.

A closed mesh constains no border. I.e. all edges have at least 2 incident faces.

is_edge_manifold()
Return true iff this mesh is edge-manifold.

A mesh is edge-manifold if there are exactly 2 incident faces for all non-border edges. Border edges, by definition, only have 1 incident face.

is_manifold()
Return true iff this mesh is both vertex-manifold and edge-manifold.

is_oriented()
Return true iff the mesh is consistently oriented.

That is all non-bonary edges must represent locally 2-manifold or intersection of 2-manifold surfaces.

is_vertex_manifold()
Return true iff this mesh is vertex-manifold.

A mesh is vertex-manifold if the 1-ring neighborhood of each vertex is a topological disk.

set_attribute(*name*, *val*)
Set attribute to the given value.

Reading and Writing Meshes

pymesh.meshio.**load_mesh**(*filename*, *drop_zero_dim=False*)
Load mesh from a file.

Parameters

- **filename** – Input filename. File format is auto detected based on extension.
- **drop_zero_dim** (*bool*) – If true, convert flat 3D mesh into 2D mesh.

Returns The loaded mesh.

Return type *Mesh*

pymesh.meshio.**save_mesh**(*filename*, *mesh*, **attributes*, ***setting*)
Save mesh to file.

Parameters

- **filename** (str) – Output file. File format is auto detected from extension.
- **mesh** (Mesh) – Mesh object.
- ***attributes** (list) – (optional) Attribute names to be saved. This field would be ignored if the output format does not support attributes (e.g. **.obj** and **.stl** files)
- ****setting** (dict) – (optional) The following keys are recognized.
  - ascii: whether to use ascii encoding, default is false.
  - use_float: store scalars as float instead of double, default is false.
  - anonymous: whether to indicate the file is generated by PyMesh.

Raises KeyError – Attributes cannot be found in mesh.

pymesh.meshio.**save_mesh_raw**(*filename*, *vertices*, *faces*, *voxels=None*, ***setting*)
Save raw mesh to file.

Parameters

- **filename** (str) – Output file. File format is auto detected from extension.
- **vertices** (numpy.ndarray) – Array of floats with size (num_vertices, dim).
- **faces** (numpy.ndarray) – Arrayof ints with size (num_faces, vertex_per_face).
- **voxels** (numpy.ndarray) – (optional) ndarray of ints with size (num_voxels, vertex_per_voxel). Use None for forming surface meshes.
- ****setting** (dict) – (optional) The following keys are recognized.
  - ascii: whether to use ascii encoding, default is false.

- use_float: store scalars as float instead of double, default is false.
- anonymous: whether to indicate the file is generated by PyMesh.

pymesh.meshio.**form_mesh**(*vertices*, *faces*, *voxels=None*)
Convert raw mesh data into a Mesh object.

Parameters

- **vertices** (*numpy.ndarray*) – ndarray of floats with size (num_vertices, dim).
- **faces** – ndarray of ints with size (num_faces, vertex_per_face).
- **voxels** – optional ndarray of ints with size (num_voxels, vertex_per_voxel). Use `None` for forming surface meshes.

Returns A Mesh object.

Geometry Processing Functions

Boolean opertions

pymesh.**boolean**(*mesh_1*, *mesh_2*, *operation*, *engine='auto'*, *with_timing=False*, *exact_mesh_file=None*)
Perform boolean operations on input meshes.

Parameters

- **mesh_1** (`Mesh`) – The first input mesh, M_1.
- **mesh_2** (`Mesh`) – The second input mesh, M_2.
- **operation** (`string`) – The name of the operation. Valid choices are:
  - `intersection`: $M_1 \cap M_2$
  - `union`: $M_1 \cup M_2$
  - `difference`: $M_1 \setminus M_2$
  - `symmetric_difference`: $(M_1 \setminus M_2) \cup (M_2 \setminus M_1)$
- **engine** (`string`) – (optional) Boolean engine name. Valid engines include:
  - `auto`: Using the default boolean engine (`igl` for 3D and `clipper` for 2D). This is the default.
  - `cork`: Cork 3D boolean libary
  - `cgal`: CGAL 3D boolean operations on Nef Polyhedra
  - `corefinement`: The undocumented CGAL boolean function that does not use Nef Polyhedra.
  - `igl`: libigl's 3D boolean support
  - `clipper`: Clipper 2D boolean library
  - `carve`: Carve solid geometry library
- **with_timing** (`boolean`) – (optional) Whether to time the code.
- **exact_mesh_file** (`str`) – (optional) Filename to store the XML serialized exact output.

Returns: The output mesh.

The following attributes are defined in the output mesh:

- "source": An array of 0s and 1s indicating which input mesh an output face comes from.
- "source_face": An array of indices, one per output face, into the concatenated faces of the input meshes.

Self-intersection

pymesh.**resolve_self_intersection**(*mesh*, *engine='auto'*)

Resolve all self-intersections.

Parameters

- **mesh** (Mesh) – The input mesh. Only triangular mesh is supported.
- **engine** (string) – (optional) Self-intersection engine. Valid engines include:
  - auto: the default engine (default is igl).
  - igl: libigl's self-intersection engine

Returns

A triangular mesh with all self-intersection meshed. The following per-face scalar field is defined:

- **face_sources: For each output face, this field specifies the index** of the corresponding "source face" in the input mesh.

Return type Mesh

pymesh.**detect_self_intersection**(*mesh*)

Detect all self-intersections.

Parameters **mesh** (Mesh) – The input mesh.

Returns

A n by 2 array of face indices. Each row contains the indices of two intersecting faces. n is the number of intersecting face pairs.

Return type numpy.ndarray

Outer Hull

pymesh.**compute_outer_hull**(*mesh*, *engine='auto'*, *all_layers=False*)

Compute the outer hull of the input mesh.

Parameters

- **engine** (str) – (optional) Outer hull engine name. Valid engines are:
  - auto: Using the default engine (igl).
  - igl: libigl's outer hull support
- **all_layers** (bool) – (optional) If true, recursively peel outer hull layers.

Returns

If all_layers is false, just return the outer hull mesh.

If `all_layers` is ture, return a recursively peeled outer hull layers, from the outer most layer to the inner most layer.

The following mesh attirbutes are defined in each outer hull mesh:

- `flipped`: A per-face attribute that is true if a face in outer hull is orientated differently comparing to its corresponding face in the input mesh.
- `face_sources`: A per-face attribute that specifies the index of the source face in the input mesh.

Distance to Mesh

`pymesh.`**`distance_to_mesh`**(*mesh*, *pts*)

Compute the distance from a set of points to a mesh.

Parameters

- **`mesh`** (`Mesh`) – A input mesh.
- **`pts`** (`numpy.ndarray`) – A $N \times dim$ array of query points.

Returns

Three values are returned.

- `squared_distances`: squared distances from each point to mesh.
- `face_indices` : the closest face to each point.
- **`closest_points`: the point on mesh that is closest to each** query point.

Triangulation

`pymesh.`**`triangulate`**(*points*, *segments*, *max_area*, *holes=None*, *split_boundary=True*, *auto_hole_detection=False*, *use_steiner_points=True*)

2D and 3D planar Delaunay triangulation.

Parameters

- **`points`** (`numpy.ndarray`) – A set of 2D or 3D planar nodes.
- **`segments`** (`numpy.ndarray`) – Segments that connect nodes.
- **`max_area`** (*`float`*) – The desired maximum output triangle area.
- **`holes`** (`numpy.ndarray`) – (optional) A set of 2D or 3D points such that the region containing any of these points are treated as holes. If set to None, assumes no hole exists unless `auto_hole_detection` flag is set.
- **`split_boundary`** (*`bool`*) – (optional) Whether splitting boundary segment is allowed.
- **`auto_hole_detection`** (*`bool`*) – (optional) Automatically determine the regions that correspond to holes.
- **`use_steiner_points`** (*`bool`*) – (optional) Whether to allow insertion of steiner points (automatically generagted points added to the interior of the mesh to improve triangle quality).

Note that `points` and `segments` should form one or more closed planar curve representing the boundaries of the 2D regions to be triangulated.

Returns The output `vertex` and `faces`.

pymesh.**retriangulate**(*mesh*, *max_area*, *split_boundary=True*, *use_steiner_points=True*)

Retriangulate a given 2D or 3D planar mesh.

Parameters

- **mesh** (Mesh) – The input 2D or 3D planar mesh.
- **max_area** (*float*) – The desired maximum output triangle area.
- **split_boundary** (*bool*) – (optional) Whether splitting boundary segment is allowed.
- **use_steiner_points** (*bool*) – (optional) Whether to allow insertion of steiner points (automatically generagted points added to the interior of the mesh to improve triangle quality).

Returns The retriangulated mesh.

pymesh.**retriangulate_raw**(*vertices*, *faces*, *max_area*, *split_boundary=True*, *use_steiner_points=True*)

Retriangulate a given 2D or 3D planar mesh.

Parameters

- **vertices** (numpy.ndarray) – The vertices of the input mesh.
- **faces** (numpy.ndarray) – The faces of the input mesh.
- **max_area** (*float*) – The desired maximum output triangle area.
- **split_boundary** (*bool*) – (optional) Whether splitting boundary segment is allowed.
- **use_steiner_points** (*bool*) – (optional) Whether to allow insertion of steiner points (automatically generagted points added to the interior of the mesh to improve triangle quality).

Returns The vertices and faces of retriangulated mesh.

Cell partitions

pymesh.**partition_into_cells**(*mesh*)

Resolve all-intersections of the input mesh and extract cell partitions induced by the mesh. A cell-partition is subset of the ambient space where any pair of points belonging the partition can be connected by a curve without ever going through any mesh faces.

Parameters **mesh** (Mesh) – The input mesh.

Returns: The output mesh with all intersections resolved and a list of meshes representing individual cells.

The following attributes are defined in the output mesh:

- source_face: the original face index.
- patches: the scalar field marking manifold patches (A set of connected faces connected by manifold edges).
- cells: a per-face scalar field indicating the cell id on the positive side of each face.
- winding_number: the scalar field indicating the piece-wise constant winding number of the cell on the positive side of each face.

Minkowski Sum

pymesh.**minkowski_sum**(*mesh*, *path*)
Perform Minkowski sum of a mesh with a poly-line.

Parameters

- **mesh** (Mesh) – Input mesh.
- **path** (numpy.ndarray) – a $nimes3$ matrix. Each row represents a node in the poly-line.

Returns: A mesh representing the Minkowski sum of the inputs.

The meshutils package

pymesh.meshutils.**collapse_short_edges**(*mesh*, *abs_threshold=0.0*, *rel_threshold=None*, *preserve_feature=False*)
Wrapper function of *collapse_short_edges_raw()*.

Parameters

- **mesh** (Mesh) – Input mesh.
- **abs_threshold** (float) – (optional) All edge with length below or equal to this threshold will be collapsed. This value is ignored if rel_thresold is not None.
- **rel_threashold** (float) – (optional) Relative edge length threshold based on average edge length. e.g. rel_threshold=0.1 means all edges with length less than 0.1 * ave_edge_length will be collapsed.
- **preserve_feature** (bool) – True if shape features should be preserved. Default is false.

Returns

2 values are returned.

- output_Mesh (Mesh): Output mesh.
- information (dict): A dict of additional informations.

The following attribute are defined:

- face_sources: The index of input source face of each output face.

The following fields are defined in information:

- num_edge_collapsed: Number of edge collapsed.

pymesh.meshutils.**collapse_short_edges_raw**(*vertices*, *faces*, *abs_threshold=0.0*, *rel_threshold=None*, *preserve_feature=False*)
Convenient function for collapsing short edges.

Parameters

- **vertices** (numpy.ndarray) – Vertex array. One vertex per row.
- **faces** (numpy.ndarray) – Face array. One face per row.
- **abs_threshold** (float) – (optional) All edge with length below or equal to this threshold will be collapsed. This value is ignored if rel_thresold is not None.
- **rel_threashold** (float) – (optional) Relative edge length threshold based on average edge length. e.g. rel_threshold=0.1 means all edges with length less than 0.1 * ave_edge_length will be collapsed.

- **preserve_feature** (bool) – True if shape features should be preserved. Default is false.

Returns

3 values are returned.

- output_vertices: Output vertex array. One vertex per row.
- output_faces: Output face array. One face per row.
- information: A dict of additional informations.

The following fields are defined in information:

- num_edge_collapsed: Number of edge collapsed.
- source_face_index: An array tracks the source of each output face. That is face i of the output_faces comes from face source_face_index[i] of the input faces.

pymesh.meshutils.**generate_box_mesh**(*box_min*, *box_max*, *num_samples=1*, *keep_symmetry=False*, *subdiv_order=0*, *using_simplex=True*)

Generate axis-aligned box mesh.

Each box is made of a number of cells (a square in 2D and cube in 3D), and each cell is made of triangles (2D) or tetrahedra (3D).

Parameters

- **box_min** (numpy.ndarray) – min corner of the box.
- **box_max** (numpy.ndarray) – max corner of the box.
- **num_samples** (int) – (optional) Number of segments on each edge of the box. Default is 1.
- **keep_symmetry** (bool) – (optional) If true, ensure mesh connectivity respect all reflective symmetries of the box. Default is true.
- **subdiv_order** (int) – (optional) The subdivision order. Default is 0.
- **using_simplex** (bool) – If true, build box using simplex elements (i.e. triangle or tets), otherwise, use quad or hex element.

Returns

The output box mesh. The following attributes are defined:

- cell_index: An numpy.ndarray of size N_e that maps each element to the index of the cell it belongs to. N_e is the number of elements.

Return type Mesh

pymesh.meshutils.**generate_dodecahedron**(*radius*, *center*)

Generate dodecahedron.

Parameters

- **radius** (float) – Radius of the shape.
- **center** (numpy.ndarray) – shape center.

Returns The dodecahedron mesh.

pymesh.meshutils.**generate_icosphere**(*radius*, *center*, *refinement_order=0*)

Generate icosphere.

Parameters

- **radius** (float) – Radius of icosphere.
- **center** (numpy.ndarray) – Sphere center.
- **refinement_order** (int) – (optional) Number of refinement.

Returns The icosphere mesh.

pymesh.meshutils.**get_degenerated_faces**(*mesh*)
A thin wrapper for *get_degenerated_faces_raw()*.

pymesh.meshutils.**get_degenerated_faces_raw**(*vertices*, *faces*)
Return indices of degenerated faces. A face is degenerated if all its 3 corners are colinear.

Parameters

- **vertices** (numpy.ndarray) – Vertex matrix.
- **faces** (numpy.ndarray) – Face matrix.

Returns List of indices of degenerated faces.

Return type numpy.ndarray

pymesh.meshutils.**hex_to_tet**(*mesh*, *keep_symmetry=False*, *subdiv_order=0*)
Convert hex mesh into tet mesh.

Parameters

- **mesh** (Mesh) – Input hex mesh.
- **keep_symmetry** (*boolean*) – (optional) Whether to split hex symmetrically into tets. Default is False.
- **subdiv_order** (*int*) – (optional) Number of times to subdiv the hex before splitting. Default is 0.

Returns The resulting tet mesh.

pymesh.meshutils.**quad_to_tri**(*mesh*, *keep_symmetry=False*)
Convert quad mesh into triangles.

Parameters

- **mesh** (Mesh) – Input quad mesh.
- **keep_symmetry** (*boolean*) – (optional) Whether to split quad symmetrically into triangles. Default is False.

Returns The resulting triangle mesh.

pymesh.meshutils.**is_colinear**(*v0*, *v1*, *v2*)
Return true if v0, v1 and v2 are colinear. Colinear check is done using exact predicates.

Parameters

- **v0** (numpy.ndarray) – vector of size 2 or 3.
- **v1** (numpy.ndarray) – vector of size 2 or 3.
- **v2** (numpy.ndarray) – vector of size 2 or 3.

Returns whether v0, v1 and v2 are colinear.

Return type bool

pymesh.meshutils.**merge_meshes**(*input_meshes*)
Merge multiple meshes into a single mesh.

Parameters **input_meshes** (list) – a list of input Mesh objects.

Returns

An mesh consists of all vertices, faces and voxels from input_meshes. The following mesh attributes are defined:

- vertex_sources: Indices of source vertices from the input mesh.
- face_sources: Indices of source faces from the input mesh if the output contains at least 1 face.
- voxel_sources: Indices of source voxels from the input mesh if the output contains at least 1 voxel.

Return type Mesh

pymesh.meshutils.**remove_degenerated_triangles**(*mesh*, *num_iterations=5*)
Wrapper function of *remove_degenerated_triangles_raw()*.

Parameters **mesh** (Mesh) – Input mesh.

Returns Output mesh without degenerated triangles. info: Additional information dictionary.

Return type Mesh

pymesh.meshutils.**remove_degenerated_triangles_raw**(*vertices*, *faces*, *num_iterations=5*)
Remove degenerated triangles.

Degenerated faces are faces with zero area. It is impossible to compute face normal for them. This method get rid of all degenerated faces. No new vertices will be introduced. Only connectivity is changed.

Parameters

- **vertices** (numpy.ndarray) – Vertex array with one vertex per row.
- **faces** (numpy.ndarray) – Face array with one triangle per row.

Returns

3 values are returned.

- output_vertices: Output vertex array, one vertex per row.
- output_faces: Output face array, one face per row.
- info: Additional information dict. The following fields are defined:
- ori_face_indices: index array that maps each output face to an input face that contains it.

pymesh.meshutils.**remove_duplicated_faces**(*mesh*, *fins_only=False*)
Wrapper function of *remove_duplicated_faces_raw()*.

Parameters

- **mesh** (Mesh) – Input mesh.
- **fins_only** (bool) – If set, only remove fins.

Returns

2 values are returned.

- output_mesh (Mesh): Output mesh.

- `information` (`dict`): A `dict` of additional informations.

The following fields are defined in `information`:

- `ori_face_index`: An array of original face indices. I.e. face `i` of the `output_faces` has index `ori_face_index[i]` in the input vertices.

`pymesh.meshutils.`**`remove_duplicated_faces_raw`**(*vertices*, *faces*, *fins_only=False*)

Remove duplicated faces.

Duplicated faces are defined as faces consist of the same set of vertices. Depending on the face orientation. A special case of duplicated faces is a fin. A fin is defined as two duplicated faces with opposite orientaiton.

If fins_only is set to True, all fins in the mesh are removed. The output mesh could still contain duplicated faces but no fins.

If fins_only is not True, all duplicated faces will be removed. There could be two caes:

If there is a dominant orientation, that is more than half of the faces are consistently orientated, and fins_only is False, one face with the dominant orientation will be kept while all other faces are removed.

If there is no dominant orientation, i.e. half of the face are positively orientated and the other half is negatively orientated, all faces are discarded.

Parameters

- **`vertices`** (`numpy.ndarray`) – Vertex array with one vertex per row.
- **`faces`** (`numpy.ndarray`) – Face array with one face per row.
- **`fins_only`** (`bool`) – If set, only remove fins.

Returns

3 values are returned.

- `output_vertices`: Output vertex array, one vertex per row.
- `output_faces`: Output face array, one face per row.
- `information`: A `dict` of additional informations.

The following fields are defined in `information`:

- `ori_face_index`: An array of original face indices. I.e. face `i` of the `output_faces` has index `ori_face_index[i]` in the input vertices.

`pymesh.meshutils.`**`remove_duplicated_vertices`**(*mesh*, *tol=1e-12*, *importance=None*)

Wrapper function of *remove_duplicated_vertices_raw()*.

Parameters

- **`mesh`** (`Mesh`) – Input mesh.
- **`tol`** (`float`) – (optional) Vertices with distance less than `tol` are considered as duplicates. Default is `1e-12`.
- **`importance`** (`numpy.ndarray`) – (optional) Per-vertex importance value. When discarding duplicates, the vertex with the highest importance value will be kept.

Returns

2 values are returned.

- `output_mesh` (`Mesh`): Output mesh.
- `information` (`dict`): A `dict` of additional informations.

The following fields are defined in `information`:

- `num_vertex_merged`: number of vertex merged.
- `index_map`: An array that maps input vertex index to output vertex index. I.e. vertex `i` will be mapped to `index_map[i]` in the output.

`pymesh.meshutils.`**`remove_duplicated_vertices_raw`**(*vertices*, *elements*, *tol=1e-12*, *importance=None*)

Merge duplicated vertices into a single vertex.

Parameters

- **`vertices`** (`numpy.ndarray`) – Vertices in row major.
- **`elements`** (`numpy.ndarray`) – Elements in row major.
- **`tol`** (`float`) – (optional) Vertices with distance less than `tol` are considered as duplicates. Default is `1e-12`.
- **`importance`** (`numpy.ndarray`) – (optional) Per-vertex importance value. When discarding duplicates, the vertex with the highest importance value will be kept.

Returns

3 values are returned.

- `output_vertices`: Output vertices in row major.
- `output_elements`: Output elements in row major.
- `information`: A `dict` of additional informations.

The following fields are defined in `information`:

- `num_vertex_merged`: number of vertex merged.
- `index_map`: An array that maps input vertex index to output vertex index. I.e. vertex `i` will be mapped to `index_map[i]` in the output.

`pymesh.meshutils.`**`remove_isolated_vertices`**(*mesh*)

Wrapper function of *remove_isolated_vertices_raw()*.

Parameters **`mesh`** (`Mesh`) – Input mesh.

Returns

2 values are returned.

- `output_mesh` (`Mesh`): Output mesh.
- `infomation` (`dict`): A `dict` of additional informations.

The following fields are defined in `infomation`:

- `num_vertex_removed`: Number of vertex removed.
- `ori_vertex_index`: Original vertex index. That is vertex `i` of `output_vertices` has index `ori_vertex_index[i]` in the input vertex array.

`pymesh.meshutils.`**`remove_isolated_vertices_raw`**(*vertices*, *elements*)

Remove isolated vertices.

Parameters

- **`vertices`** (`numpy.ndarray`) – Vertex array with one vertex per row.
- **`elements`** (`numpy.ndarray`) – Element array with one face per row.

Returns

3 values are returned.

- `output_vertices`: Output vertex array with one vertex per row.
- `output_elements`: Output element array with one element per row.
- `infomation`: A `dict` of additional informations.

The following fields are defined in `infomation`:

- `num_vertex_removed`: Number of vertex removed.
- `ori_vertex_index`: Original vertex index. That is vertex `i` of `output_vertices` has index `ori_vertex_index[i]` in the input vertex array.

`pymesh.meshutils.`**`remove_obtuse_triangles`**(*mesh*, *max_angle=120*, *max_iterations=5*)

Wrapper function of *`remove_obtuse_triangles_raw()`*.

Parameters

- **`mesh`** (`Mesh`) – Input mesh.
- **`max_angle`** (`float`) – (optional) Maximum obtuse angle in degrees allowed. All triangle with larger internal angle would be split. Default is 120 degrees.
- **`max_iterations`** (`int`) – (optional) Number of iterations to run before quitting. Default is 5.

Returns

2 values are returned.

- `output_mesh` (`Mesh`): Output mesh.
- `information` (`dict`): A `dict` of additinal informations.

The following fields are defiend in `information`:

- `num_triangle_split`: number of triangles split.

`pymesh.meshutils.`**`remove_obtuse_triangles_raw`**(*vertices*, *faces*, *max_angle=120*, *max_iterations=5*)

Remove all obtuse triangles.

Parameters

- **`vetices`** (`numpy.ndarray`) – Vertex array with one vertex per row.
- **`faces`** (`numpy.ndarray`) – Face array with one face per row.
- **`max_angle`** (`float`) – (optional) Maximum obtuse angle in degrees allowed. All triangle with larger internal angle would be split. Default is 120 degrees.
- **`max_iterations`** (`int`) – (optional) Number of iterations to run before quitting. Default is 5.

Returns

3 values are returned.

- `output_vertices`: Output vertex array with one vertex per row.
- `output_faces`: Output face array with one face per row.
- `information`: A `dict` of additinal informations.

The following fields are defiend in `information`:

- num_triangle_split: number of triangles split.

pymesh.meshutils.**separate_mesh**(*mesh*, *connectivity_type='auto'*)
Split mesh into connected components.

Parameters

- **mesh** (Mesh) – Input mesh.
- **connectivity_type** (str) – possible types are
  - auto: Same as face for surface mesh, voxel for voxel mesh.
  - vertex: Group component based on vertex connectivity.
  - face: Group component based on face connectivity.
  - voxel: Group component based on voxel connectivity.

Returns

A list of meshes, each represent a single connected component. Each output component have the following attributes defined:

- ori_vertex_index: The input vertex index of each output vertex.
- ori_elem_index: The input element index of each output element.

pymesh.meshutils.**separate_graph**(*edges*)
Split graph into disconnected components.

Parameters **edges** (numpy.ndarray) – edges of the graph.

Returns An array of indices indicating the component each edge belongs to.

pymesh.meshutils.**split_long_edges**(*mesh*, *max_edge_length*)
Wrapper function of *split_long_edges_raw()*.

Parameters

- **mesh** (Mesh) – Input mesh.
- **max_edge_length** (float) – Maximum edge length allowed. All edges longer than this will be split.

Returns

2 values are returned.

- output_mesh (Mesh): Output mesh.
- information: A dummy dict that is currently empty. It is here to ensure consistent interface across the module.

pymesh.meshutils.**split_long_edges_raw**(*vertices*, *faces*, *max_edge_length*)
Split long edges.

Parameters

- **vertices** (numpy.ndarray) – Vertex array with one vertex per row.
- **faces** (numpy.ndarray) – Face array with one face per row.
- **max_edge_length** (float) – Maximum edge length allowed. All edges longer than this will be split.

Returns

3 values are returned.

- `output_vertices`: Output vertex array with one vertex per row.
- `output_faces`: Output face array with one face per row.
- `information`: A dummy `dict` that is currently empty. It is here to ensure consistent interface across the module.

`pymesh.meshutils.`**`subdivide`**(*mesh*, *order=1*, *method='simple'*)

Subdivide the input mesh.

Parameters

- **`mesh`** – Input triangle mesh.
- **`order`** – (optional) Subdivision order.
- **`method`** – (optional) Subdivision method. Choices are "simple" and "loop".

Returns Returns the subdivided mesh. The per-face attribute "ori_face_index" tracks the original face index from the input mesh.

The `misc` package

class `pymesh.misc.`**`Quaternion`**(*quat=[1, 0, 0, 0]*)

This class implements quaternion used for 3D rotations.

`w`

`float` – same as `quaternion[0]`.

`x`

`float` – same as `quaternion[1]`.

`y`

`float` – same as `quaternion[1]`.

`z`

`float` – same as `quaternion[2]`.

`conjugate`()

returns the conjugate of this quaternion, does nothing to self.

classmethod `fromAxisAngle`(*axis*, *angle*)

Crate quaternion from axis angle representation

Parameters

- **`angle`** (`float`) – Angle in radian.
- **`axis`** (`numpy.ndarray`) – Rotational axis. Not necessarily normalized.

Returns

The following values are returned.

- `quat` (*`Quaternion`*): The corresponding quaternion object.

classmethod `fromData`(*v1*, *v2*)

Create the rotation to rotate v1 to v2

Parameters

- **`v1`** (`numpy.ndarray`) – Vrom vector. Normalization not necessary.
- **`v2`** (`numpy.ndarray`) – To vector. Normalization not necessary.

Returns

The following values are returned.

- `quat` (*Quaternion*): Corresponding quaternion that rotates `v1` to `v2`.

norm()
Quaternion norm.

normalize()
Normalize quaterion to have length 1.

rotate(*v*)
Rotate 3D vector v by this quaternion

Parameters **v** (`numpy.ndarray`) – Must be 1D vector.

Returns The rotated vector.

to_matrix()
Convert to rotational matrix.

Returns The corresponding rotational matrix.

Return type `numpy.ndarray`

Tiling and Inflating Wires

class `pymesh.wires.`**WireNetwork**
Data structure for wire network.

A wire network consists of a list of vertices and a list of edges. Thus, it is very similar to a graph except all vertices have their positions specified. Optional, each vertex and edge could have multiple attributes.

compute_symmetry_orbits()
Compute the following symmetry orbits:

- `vertex_symmetry_orbit`: all vertices belonging to the same orbit can be mapped to each other by reflection with respect to planes orthogonal to axis.
- `vertex_cubic_symmetry_orbit`: all vertices belonging to the same orbit can be mapped to each other by reflection with respect to any symmetry planes of a perfect cube.
- `edge_symmetry_orbit`: all edges belonging to the same orbit can be mapped to each other by reflection with respect to planes orthogonal to axis.
- `edge_cubic_symmetry_orbit`: all edges belonging to the same orbit can be mapped to each other by reflection with respect to any symmetry planes of a perfect cube.

All orbits are stored as attributes.

dim
Dimension of the ambient space, choices are 2 or 3.

filter_edges(*to_keep*)
Remove all edges unless marked with to keep. Vertices are left unchanged.

filter_vertices(*to_keep*)
Remove all vertices other than the ones marked with to_keep. Edges are updated accordingly.

load(*vertices*, *edges*)
Load vertices and edges from data.

Parameters

- **vertices** (numpy.ndarray) – num_vertices by dim array of vertex coordinates.
- **faces** (numpy.ndarray) – num_edges by 2 array of vertex indices.

load_from_file(*wire_file*)
Load vertices and edges from a file.

Parameters **wire_file** (str) – Input wire file name.

The file should have the following format:

```
# This is a comment
v x y z
v x y z
...
l i j # where i and j are vertex indices (starting from 1)
l i j
...
```

load_from_raw(*raw_wires*)
Load vertex and edges from raw C++ wire data structure.

offset(*offset_vector*)
Offset vertices by per-vertex offset_vector.

Parameters **offset_vector** (numpy.ndarray) – A $Nimesdim$ matrix representing per-vertex offset vectors.

scale(*factors*)
Scale the wire network by factors

Parameters **factors** – scaling factors. Scale uniformly if factors is a scalar. If factors is an array, scale each dimension separately (dimension i is scaled by factors[i]).

trim()
Remove all hanging edges. e.g. edge with at least one vertex of valance <= 1

class pymesh.wires.**Parameters**(*wire_network*, *default_thickness=0.5*)
This class is a thin wrapper around PyWires.ParameterManager class.

class pymesh.wires.**Tiler**(*base_pattern=None*)

class pymesh.wires.**Inflator**(*wire_network*)

set_profile(*N*)
Set the cross section shape of each wire to N-gon.

set_refinement(*order=1*, *method='loop'*)
Refine the output mesh using subdivision.

Parameters

- **order** – how many times to subdivide.
- **mehtod** – which subdivision scheme to use. Options are loop and simple.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index