PyMesh Documentation

Release 0.2.1

Qingnan Zhou

Dec 23, 2018

Contents

1 Features: **3**

2 Contents: **5**

2.1 User Guide . . . 5
2.1.1 Installation . . . 5
2.1.2 Basic Usage . . . 8
2.1.3 Mesh Processing . . . 12
2.1.4 Mesh Boolean . . . 17
2.1.5 Wire Inflation . . . 20
2.2 PyMesh API Reference . . . 28
2.2.1 Mesh Data Structure . . . 28
2.2.2 Reading and Writing Meshes . . . 30
2.2.3 Local Mesh Cleanup . . . 32
2.2.4 Procedural Mesh Generation . . . 40
2.2.5 Mesh Generation . . . 45
2.2.6 Geometry Processing Functions . . . 49
2.2.7 Finite Element Matrix Assembly . . . 54
2.2.8 Sparse Linear System Solver . . . 54
2.2.9 Miscellaneous functions . . . 56

3 Indices and tables **63**

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, TetGen and Quartet.
- Wire network and inflation of wire networks.
- Finite element matrix assembly. (supports Laplacian, stiffness, mass, etc.)

CHAPTER 2

Contents:

2.1 User Guide

2.1.1 Installation

Docker

The easiest way of using PyMesh is through docker, where one can simply `pull` a prebuild image of PyMesh from dockerhub:

```
$ docker run -it pymesh/pymesh
Python 3.6.4 (default, Dec 21 2017, 01:35:12)
[GCC 4.9.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pymesh
>>>
```

Download the Source

The source code can be checked out from GitHub:

```
git clone https://github.com/PyMesh/PyMesh.git
cd PyMesh
git submodule update --init
export PYMESH_PATH=`pwd`
```

The rest of the document assumes PyMesh is located at `$PYMESH_PATH`.

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:

System dependencies

- Python: v2.7 or higher
- NumPy: v1.8 or higher
- SciPy: v0.13 or higher
- Eigen: v3.2 or higher
- TBB: 2018 Update 1 or later
- GMP: v6.1 or higher
- MPFR: v4.0 or higher
- Boost: 1.6 or higher (thread, system)

On Linux, the system dependencies can be installed with *apt-get*:

```
apt-get install \
    libeigen3-dev \
    libgmp-dev \
    libgmpxx4ldbl \
    libmpfr-dev \
    libboost-dev \
    libboost-thread-dev \
    libtbb-dev \
    python3-dev
```

On MacOS, the system dependencies can be installed with MacPorts:

```
port install
    python36 \
    eigen3 \
    gmp \
    mpfr \
    tbb \
    boost
```

Python dependencies such as NumPy and SciPy can be installed using *pip*:

```
pip install -r $PYMESH_PATH/python/requirements.txt
```

Third party dependencies

The following third-party libraries are not required, but highly recommended in order to use the full power of Pymesh. 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 software for your OS. A copy of these libraries are also included in the `third_party` directory.

- 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.
- 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.
- MMG3D: is used as a 3D tetrahedralizaiton optimization engine.

All third party libraries are attached to the repo as submodules. They are built as part of PyMesh automatically. See *Building PyMesh* section for more instructions.

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

Build with Setuptools

Setuptools builds both the main PyMesh module as well as all third party dependencies. To build PyMesh:

```
./setup.py build
```

Build with CMake

If you are familiar with C++ and CMake, there is an alternative way of building PyMesh. First compile and install all of the third party dependencies:

```
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 ..
```

PyMesh consists of several modules. To build all modules and their corresponding unit tests:

```
make
make tests
```

PyMesh libraries are all located in `$PYMESH_PATH/python/pymesh/lib` directory.

Install PyMesh

To install PyMesh in your system:

```
./setup.py install  # May require root privilege
```

Alternatively, one can install PyMesh locally:

```
./setup.py install --user
```

Post-installation check

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

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

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

2.1.2 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 generalized voxels (i.e. a volume element such as tetrahedron or hexahedron). The dimension 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 arbitrary 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 predefined:

- `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 fields, 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)
```

2.1.3 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"]
```

(continues on next page)

(continued from previous page)

```
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.

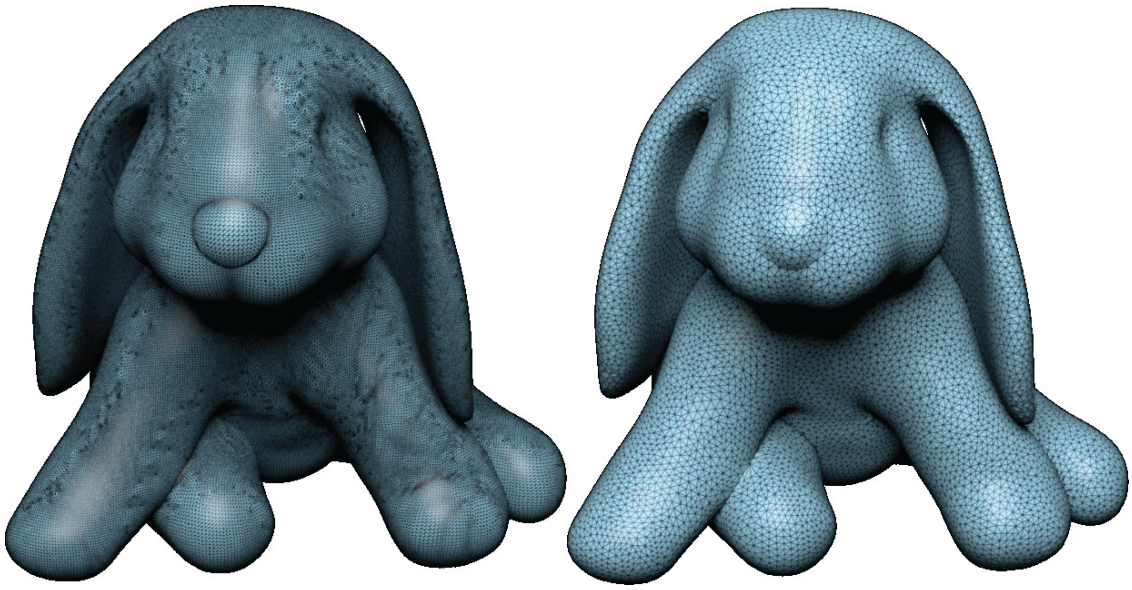
It is also possible to operate on a raw mesh:

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

This method is often used to eliminate 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']
```

(continues on next page)

(continued from previous page)

```
>>> 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 close by 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:

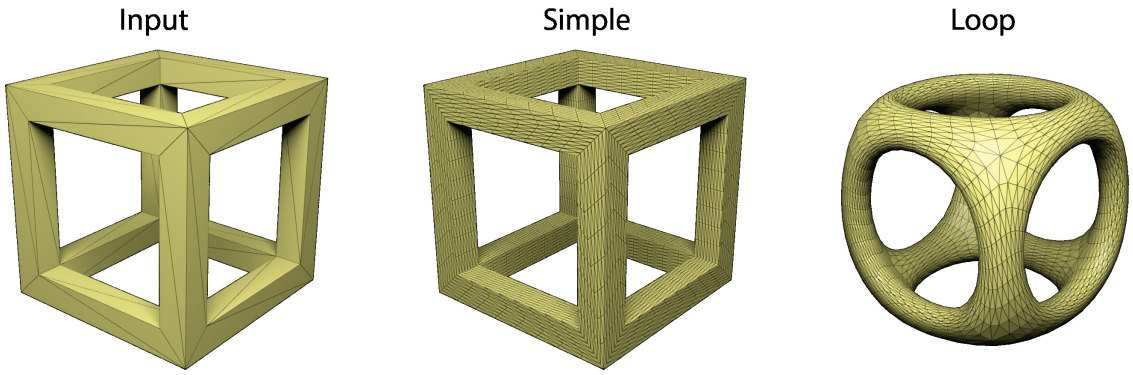
```
>>> vertices, faces, info = pymesh.remove_duplicated_faces(
...     vertices, faces)
```

Subdividing Mesh

PyMesh supports both simple and loop subdivision of a given triangular mesh:

```
>>> mesh = pymesh.subdivide(mesh, order=2, method="loop")
>>> mesh.get_attribute("ori_face_index")
array([ 0.,  0.,  0., ..., 95., 95., 95.])
```

Here are some examples of different subdivision methods:

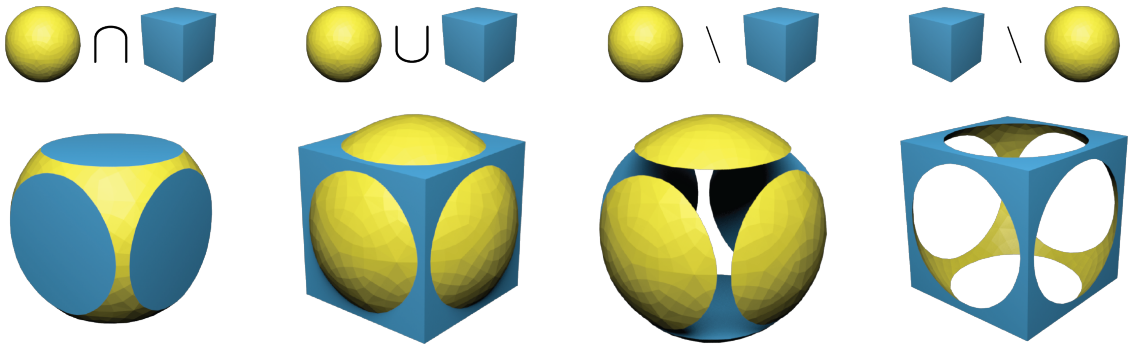


2.1.4 Mesh Boolean

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 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:

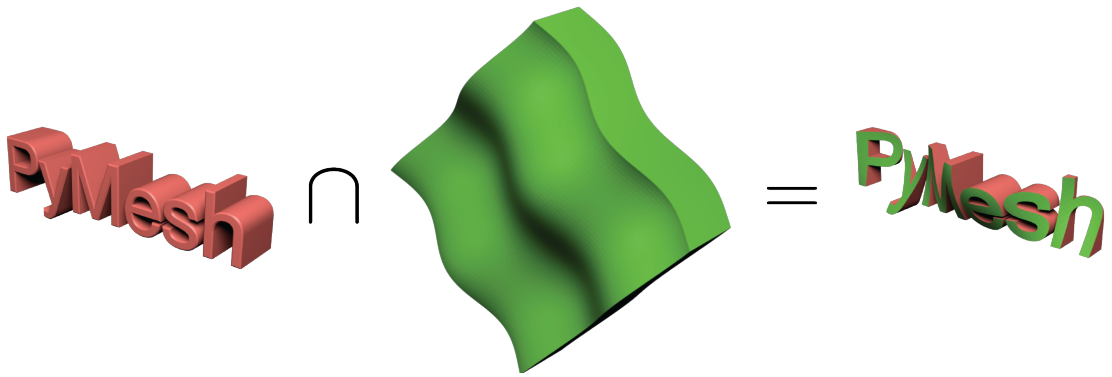
- Mesh A (pymesh.ply) contains the extruded text `PyMesh`.
- Mesh B (plate.ply) contains an extruded wavy plate.

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.])
```

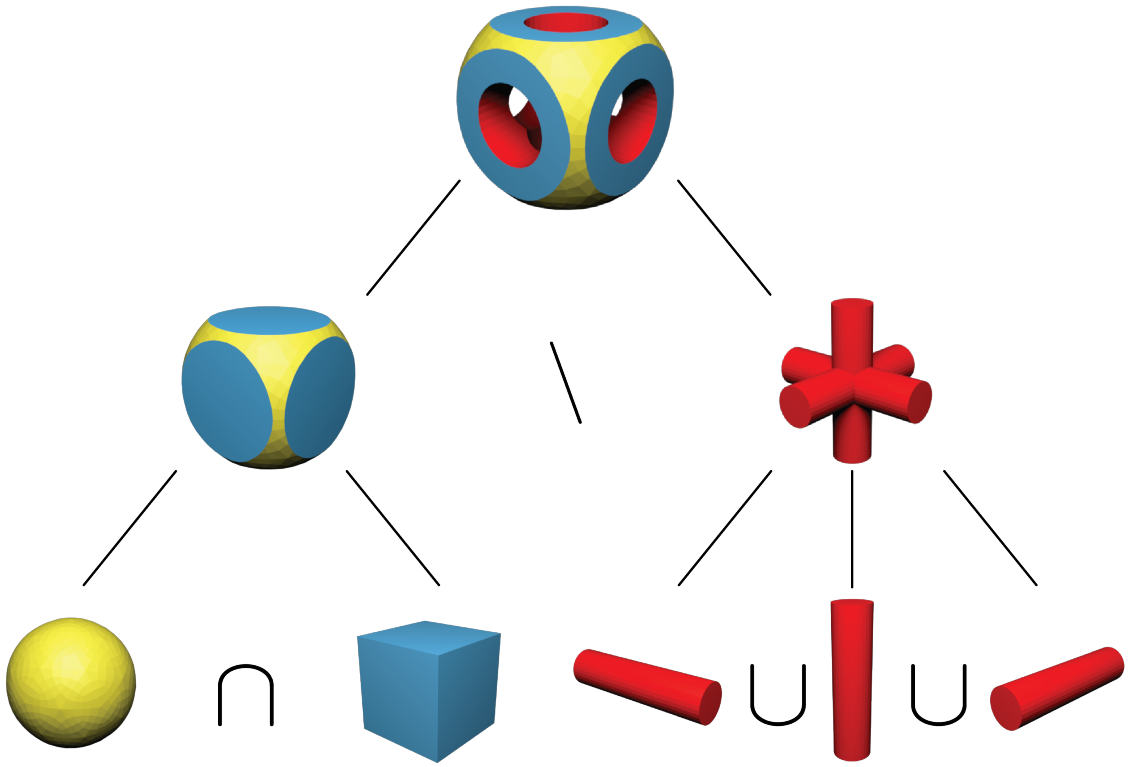
The operation is illustrated in the following figure:



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
```

(continues on next page)

(continued from previous page)

```
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
```

2.1.5 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 explicitly 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:

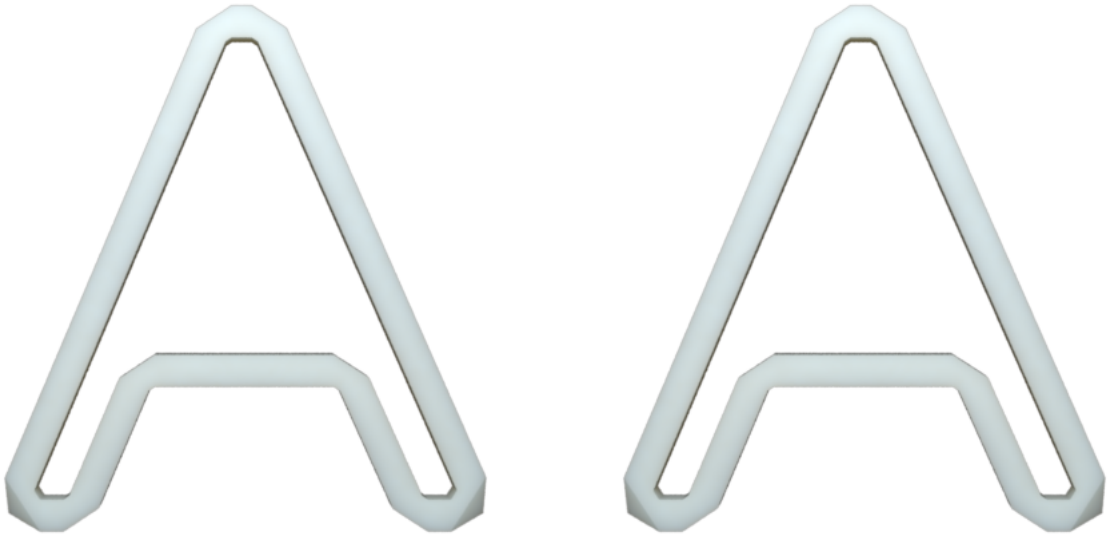


Fig. 1: Inflation output for uniform vertex thickness (left) and uniform edge thickness (right).

```
>>> # 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:

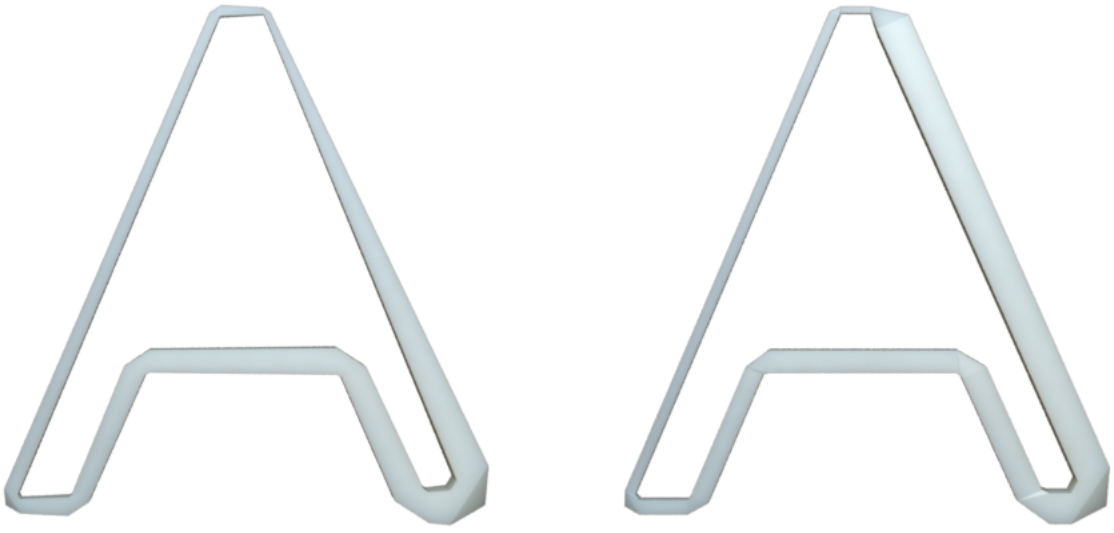


Fig. 2: Inflation output for per-vertex (left) and per-edge (right) thickness.

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:

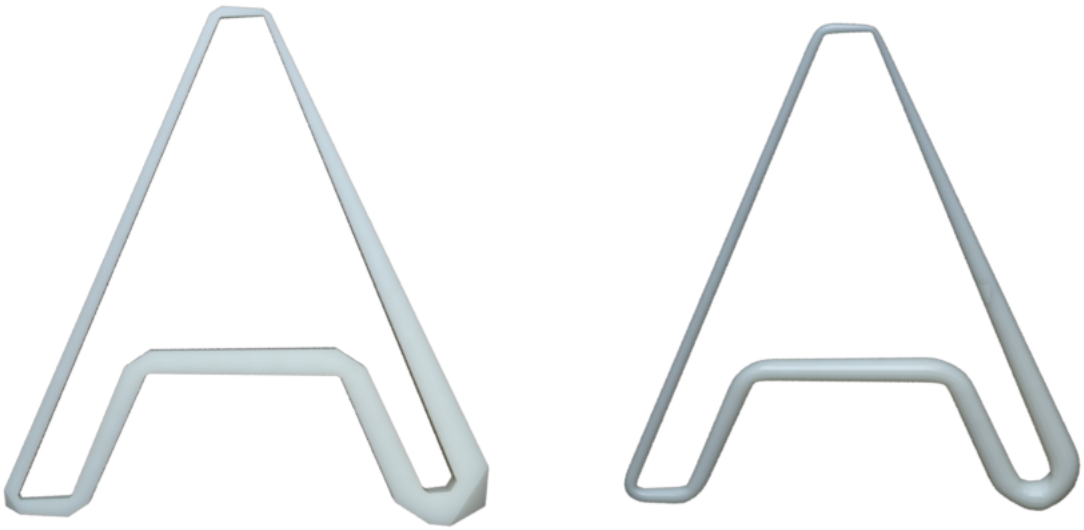


Fig. 3: Inflation output without (left) and with (right) `loop` refinement.

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 inflated. There are several ways to perform tiling.

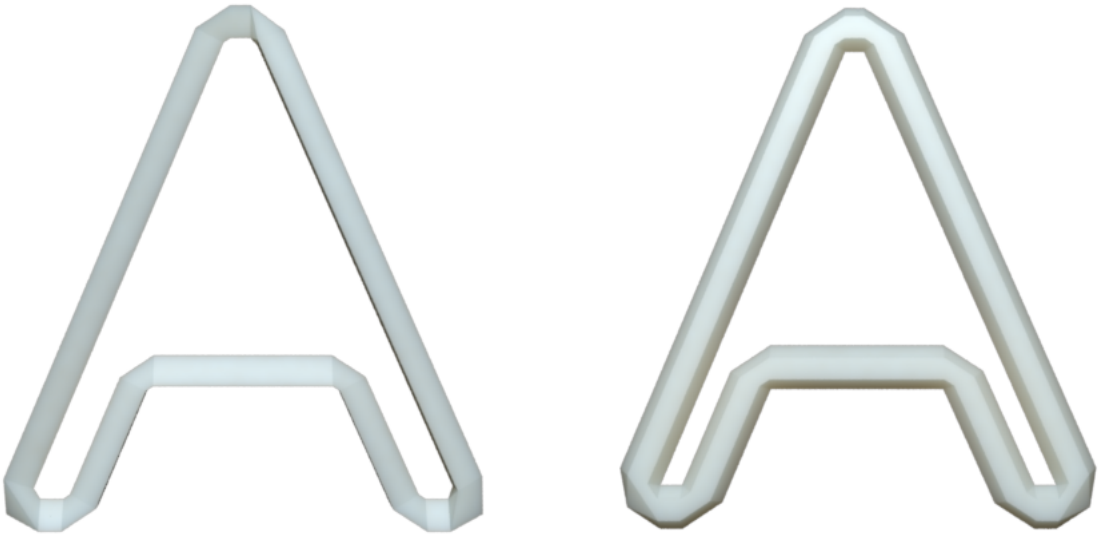


Fig. 4: Inflation with triangle profile (left) and hexagon profile (right).

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 Rubik's 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:

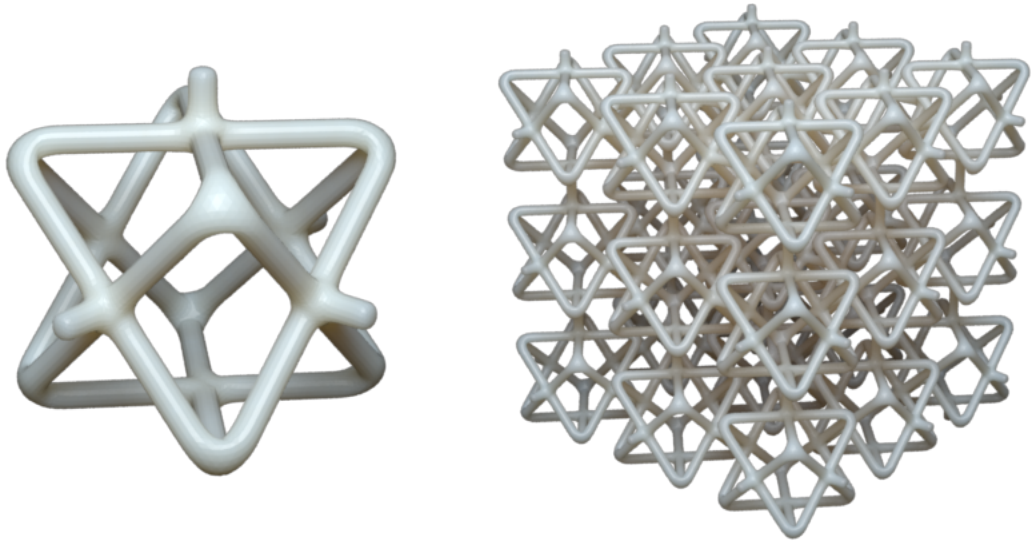


Fig. 5: A single cell wire network (left) and the corresponding 3x3x3 tiling (right).

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:

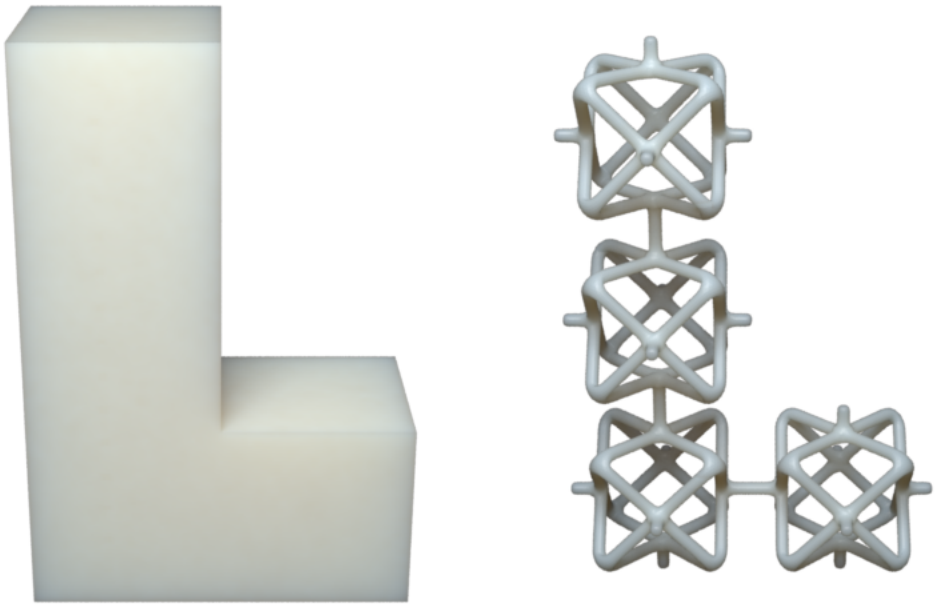


Fig. 6: Guide hex mesh (left) and the tiled result (right).

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.

2.2 PyMesh API Reference

2.2.1 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.

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

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

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

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

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

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

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

remove_attribute(*name*)
Remove attribute from mesh.

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

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

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.

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_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_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.

2.2.2 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 A Mesh object representing the loaded 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.

Example

```
>>> box = pymesh.generate_box_mesh();
>>> pymesh.save_mesh("tmp.stl", box, ascii=True);
```

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.

Example

```
>>> mesh = pymesh.generate_regular_tetrahedron();
>>> pymesh.save_mesh_raw("out.msh",
...         mesh.vertices, mesh.faces, mesh.voxels);
```

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 formed by the inputs.

Example

```
>>> vertices = np.array([
...     [0.0, 0.0],
...     [1.0, 0.0],
...     [1.0, 1.0],
...     [0.0, 1.0],
...     ]);
>>> faces = np.array([
...     [0, 1, 2],
...     [0, 2, 3],
...     ]);
>>> mesh = pymesh.form_mesh(vertices, faces);
```

2.2.3 Local Mesh Cleanup

Meshes coming from the real world are rarely clean. Artifacts such as degeneracies, duplicate vertex/triangles and self-intersections are rampant (See the `about` page in our Thingi10K dataset). Unfortunately, many geometry processing operations have strict and often unspecified requirements on the cleanness of the input geometry. Here, we provide a number of handy routines to facilitate the task of cleaning up a mesh.

Remove isolated vertices

Isolated vertices are vertices not referred by any face or voxel. They often does not contribute to the geometry (except for the case of point cloud), and thus can be safely removed.

`pymesh.`**`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.`**`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.

Remove duplicate vertices

Duplicate or near duplicate vertices are vertices with nearly same coordinates. Two vertices can be considered as duplicates of each other if their Euclidean distance is less than a tolerance (labeled *tol*). Duplicate vertices can often be merged into a single vertex.

`pymesh.`**`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.`**`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.

Collapse short edges

Short edges are edges with length less than a user specified threshold. Use the following functions to collapse *all* short edges in the mesh. The output mesh guarantees to have no short edges.

`pymesh.`**`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.`**`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.

Split long edges

Long edges are sometimes undesirable as well. Use the following functions to split long edges into 2 or more shorter edges. The output mesh guarantees to have no edges longer than the user specified threshold.

pymesh.**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.**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.

Remove duplicate faces

Duplicate faces are faces consisting of the same vertices. They are often not desirable and may cause numerical problems. Use the following functions to remove them.

pymesh.**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.**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.

Remove obtuse triangles

Obtuse triangles are often not desirable due to their geometric nature (e.g. circumcenter of obtuse triangles are outside of the triangle). Each obtuse triangle can always be split into 2 or more right or sharp triangles. They can be removed using the following routines.

pymesh.**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.**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.

Remove degenerate triangles

Degenerate triangles are triangles with collinear vertices. They have zero area and their normals are undefined. They can be removed using the following routines.

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

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

Returns

2 values are returned.

- mesh: A *Mesh* object without degenerated triangles.
- info: Additional information dictionary.

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

Degenerated faces are faces with collinear vertices. 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 in the info dict:

- ori_face_indices: index array that maps each output face to an input face that contains it.

Self-intersections

Self-intersections are often a problem in geometry processing. They can be detected and resolved with the following routines.

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

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 resolved. 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.

Separate mesh into disconnected components

`pymesh.`**`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.

Merge multiple meshes

`pymesh.`**`merge_meshes`**(*input_meshes*)

Merge multiple meshes into a single mesh.

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

Returns

A *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.

Submesh extraction

Sometimes, it is useful to extract a local region from a more complex mesh for futher examination. *pymesh.submesh()* is designed for this task.

pymesh.**submesh**(*mesh*, *element_indices*, *num_rings*)

Extract a subset of the mesh elements and forming a new mesh.

Parameters

- **mesh** (*Mesh*) – The input mesh.
- **element_indices** (numpy.ndarray) – The indices of selected elements (faces/voxels).
- **num_rings** (*int*) – The number of rings around the selected elements to extract.

Returns

A *Mesh* object only containing the selected elements and their local neighborhood up to *num_rings* rings. The output mesh contains the following attributes:

- ori_face_index/ori_voxel_index: The original index of each element.
- ring: Index indicating which ring does each element belongs. The selected elements belongs to the 0-ring.

2.2.4 Procedural Mesh Generation

Box generation

pymesh.**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

A 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.

Sphere generation

pymesh.**generate_icosphere**(*radius*, *center*, *refinement_order=0*)
Generate icosphere (subdivision surface of a regular icosahedron).

Parameters

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

Returns The (possibly refined) icosphere *Mesh*.

Cylinder generation

pymesh.**generate_cylinder**(*p0*, *p1*, *r0*, *r1*, *num_segments=16*)
Generate cylinder (or conical frustum to be precise).

Parameters

- **p0** (np.ndarray) – Bottom center.
- **p1** (np.ndarray) – Top center.
- **r0** (float) – Bottom radius.
- **r1** (float) – Top radius.
- **num_segments** (int) – Number of segments to a discrete circle consists.

Returns The cylinder *Mesh*.

Tube generation

pymesh.**generate_tube**(*p0*, *p1*, *r0_out*, *r1_out*, *r0_in*, *r1_in*, *num_segments=16*, *with_quad=False*)
Generate generalized tube (i.e. cylinder with an axial hole).

Parameters

- **p0** (np.ndarray) – Bottom center.
- **p1** (np.ndarray) – Top center.
- **r0_out** (float) – Bottom outer radius.
- **r1_out** (float) – Top outer radius.
- **r0_in** (float) – Bottom inner radius.
- **r1_in** (float) – Top inner radius.
- **num_segments** (int) – Number of segments to a discrete circle consists.
- **with_quad** (bool) – Output a quad mesh instead.

Returns A generalized tube *Mesh*.

Other Platonic solids

pymesh.**generate_regular_tetrahedron**(*edge_length=1.0, center=[0.0, 0.0, 0.0]*)
Generate a regular tetrahedron with the specified edge length and center.

Parameters

- **edge_length** (float) – edge length of the regular tet.
- **center** (numpy.ndarray) – center of the tet.

Returns A *Mesh* object containing a regular tetrahedron.

pymesh.**generate_dodecahedron**(*radius*, *center*)
Generate a regular dodecahedron.

Parameters

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

Returns The dodecahedron *Mesh* object.

Wire mesh generation

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. Optionally, each vertex and edge could be associated with one or more attributes.

dim
int – The dimension of the embedding space of this wire network. Must be 2 or 3.

vertices
numpy.ndarray – A V by dim vertex matrix. One vertex per row.

edges
numpy.ndarray – A E by 2 edge matrix. One edge per row.

num_vertices
int – The number of vertices (i.e. V).

num_edges
int – The number of edges (i.e. E).

bbox
numpy.ndarray – A 2 by dim matrix with first and second rows being the minimum and maximum corners of the bounding box respectively.

bbox_center
numpy.ndarray – Center of the bbox.

centroid
numpy.ndarray – Average of all vertices.

wire_lengths
numpy.ndarray – An array of lengths of all edges.

total_wire_length
float – Sum of wire_lengths.

attribute_names
`list` of `str` – The names of all defined attributes.

classmethod create_empty()
Handy factory method to create an empty wire network.

classmethod create_from_file(*wire_file*)
Handy factory method to load data from a file.

classmethod create_from_data(*vertices*, *edges*)
Handy factory method to create wire network from `vertices` and `edges`.

Example

```
>>> vertices = np.array([
...     [0.0, 0.0, 0.0],
...     [1.0, 0.0, 0.0],
...     [1.0, 1.0, 0.0],
...     ]);
>>> edges = np.array([
...     [0, 1],
...     [0, 2],
...     [1, 2],
...     ]);
>>> wires = WireNetwork.create_from_data(vertices, edges);
```

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.

write_to_file(*filename*)
Save the current wire network into a file.

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]`).

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

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

center_at_origin()
Translate the wire networks to have its center at the origin.

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

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

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

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.

get_attribute_names()
Get the names of all defined attributes.

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

add_attribute(*name*, *value=None*, *vertex_wise=True*)
Add a new attirbute.

Parameters

- **name** (`str`) – Attribute name.
- **value** (`numpy.ndarray`) – N by d matrix of attribute values (one row per vertex or per edge). Default is None to represent uninitialized attribute value.
- **vertex_wise** (`bool`) – Whether this attribute is assigned to vertices or edges.

get_attribute(*name*)
Get the value of an attribute.

is_vertex_attribute(*name*)
Returns true if `name` is a per-vertex attribute.

set_attribute(*name*, *value*)
Set the value of the attribute `name` to be `value`.

get_vertex_neighbors(*i*)
Returns a list of vertex indices which are connected to vertex `i` via an edge.

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

2.2.5 Mesh Generation

Triangulation

Triangulation is 2D is often solved using Shewchuk's triangle library. It is both robust and flexible. We provide a pythonic wrapper over Shewchuk's triangle that exposes most of its powers.

class pymesh.**triangle**
Wrapper around Shewchuk's triangle.

points
`numpy.ndarray` – 3D or 2D points to be triangulated. If points are embedded in 3D, they must be coplanar.

segments
`numpy.ndarray` – n by 2 matrix of indices into points. together `points` and `segments` defines a Planar Straight Line Graph (PSLG) that triangles accepts.

triangles
`numpy.ndarray` – m by 3 matrix of indices into points. When `segments` is empty and `triangles` is non-empty, use triangle to refine the existing triangulation.

holes
`numpy.ndarray` – h by dim matrix of points representing hole points. Alternatively, one can set `auto_hole_detection` to True to infer holes from the input PSLG's orientation.

min_angle
`float` – Lower bound on angle in degrees. Default is 20 degress. Setting `min_angle` > 20.7 will loose the theoretical guarentee of termination, although it often works fine in practice. However, setting `min_angle` > 34 tends to cause triangle to not terminate in practice.

max_area
`float` – Max triangle area. Default is unbounded.

max_areas
`numpy.ndarray` – Max area scalar field. It should have the same length as triangles. Not used by default.

keep_convex_hull
boolean – Whether to keep all triangles inside of the convex hull. Default is false.

conforming_delaunay
boolean – Whether to enforce conforming Delaunay triangulation. Default is false (i.e. use constrained Delaunay triangulation).

exact_arithmetic
boolean – Whether to use exact predicates. Defeault is true.

split_boundary
boolean – Whether to allow boundary to be split. Default is false.

max_num_steiner_points
int – The maximum number of Steiner points. Default is -1 (i.e. unbounded).

verbosity
int – How much info should triangle output?

0. no output
1. normal level of output
2. verbose output
3. vertex-by-vertex details
4. you must be debugging the triangle code

algorithm
str – The Delaunay triangulation algorithm to use. Choices are:

- DIVIDE_AND_CONQUER: Default. Implementation of[1].
- SWEEPLINE: Fortune's sweep line algorithm[2].
- INCREMENTAL: Also from[1].

auto_hole_detection
boolean – Whether to detect holes based on the orientation of PSLG using winding number. Default is False.

vertices
numpy.ndarray – Vertices of the output triangulation.

faces
numpy.ndarray – Faces of the output triangulation.

mesh
Mesh – Output mesh.

voronoi_vertices
numpy.ndarray – Vertices of the output Voronoi diagram. Only generated when no input segments and triangles are provided.

voronoi_edges
numpy.ndarray – Voronoi edges. Negative index indicates infinity.

regions
numpy.ndarray – Per-triangle index of connected regions separated by segments.

[1] Leonidas J. Guibas and Jorge Stolfi, Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams, ACM Transactions on Graphics 4(2):74-123, April 1985.
[2] Steven Fortune, A Sweepline Algorithm for Voronoi Diagrams, Algorithmica 2(2):153-174, 1987.

Example

```
>>> vertices = np.array([
...     [0.0, 0.0],
...     [1.0, 0.0],
...     [1.0, 1.0],
...     [0.0, 1.0],
...     ]);
>>> tri = pymesh.triangle();
>>> tri.points = vertices;
>>> tri.max_area = 0.05;
>>> tri.split_boundary = False;
>>> tri.verbosity = 0;
>>> tri.run(); # Execute triangle.
>>> mesh = tri.mesh; # output triangulation.
```

References:

Tetrahedralization

In contrast with 2D, tetrahedralization in 3D is a much hard problem. Many algorithms tries to tackle this problem from different angles. No single algorithm or package standouts as the best. We therefore offer a number of different engines for our users.

pymesh.**tetrahedralize**(*mesh*, *cell_size*, *radius_edge_ratio=2.0*, *facet_distance=-1.0*, *feature_angle=120*, *engine='auto'*, *with_timing=False*)

Create a tetrahedral mesh from input triangle mesh.

Parameters

- **mesh** (*Mesh*) – Input triangular mesh.
- **cell_size** (float) – Max radius of the circumscribed sphere of the output tet.
- **radius_edge_ratio** (float) – Max radius of the circumscribed sphere to the shortest edge length of each tet.
- **facet_distance** (float) – Upper bound on the distance from the circumcenter of a facet to the center of its "Delaunay ball", where a Delaunay ball is defined as the smallest circumscribed sphere with center on the surface of the domain.
- **feature_angle** (float) – Angle threshold (in degrees) for feature extraction.
- **engine** (string) – The tetrahedralization engine to use. Valid options are:
  - auto: default to tetgen
  - cgal: CGAL 3D mesh generation, using Polyhedron domain with auto feature extraction.
  - cgal_no_features: CGAL 3D mesh generation, using Polyhedron domain without feature extraction.
  - cgal_implicit: CGAL 3D mesh generation, using implicit domain with winding number as oracle.
  - tetgen: TetGen from Hang Si.
  - quartet: Quartet from Robert Bridson and Crawford Doran
  - delpsc: DelPSC from Tamal K Dey , Joshua A. Levine, Andrew Slatton

- `vegafem`: Tet mesher provided by VegaFEM library.
- `mmg`: Implicit domain meshing from MMG3D.
- `tetwild`: TetWild engine based on our Siggraph paper.

• **with_timing** (`boolean`) – whether to output timing info.

Returns Tetrahedral mesh (and running time if *with_timing* is True).

In addition to `pymesh.tetraheralize()`, we also provide a more complete wrapper around Si's awesome TetGen package.

class `pymesh.tetgen`

Wrapper around Si's TetGen.

All attributes, except `vertices`, `faces`, `voxels` and `mesh`, are either input geometry or configuration parameters.

points
`numpy.ndarray` – n by 3 list of points to be tetrahedralized.

triangles
`numpy.ndarray` – m by 3 matrix of indices into points. Together, points and triangles defined PLC.

tetrhaedra
`numpy.ndarray` – t by 4 matrix of indices into points. Used for refinement.

point_markers
`numpy.ndarray` – List of integer point markers of size n. Point marker cannot be 0.

point_weights
`numpy.ndarray` – List of point weights. Used for weight Delaunay tetrahedralization.

triangle_marker
`numpy.ndarray` – List of integer triangle markers of size t.

split_boundary
`bool` – whether to split input boundary. Default is true.

max_radius_edge_ratio
`float` – Default is 2.0.

min_dihedral_angle
`float` – Default is 0.0.

coarsening
`bool` – Coarsening the input tet mesh. Default is false.

max_tet_volume
`float` – Default is unbounded.

optimization_level
`int` – Ranges from 0 to 10. Default is 2.

max_num_steiner_points
`int` – Default is unbounded.

coplanar_tolerance
`float` – Used for determine when 4 points are coplanar. Default is 1e-8.

exact_arithmetic
`bool` – Whether to use exact predicates. Default is true.

merge_coplanar
`bool` – Whether to merge coplanar faces and nearby vertices. Default is true.

weighted_delaunay
`bool` – Compute weighted Delaunay tetrahedralization instead of conforming Delaunay. Default is false. This option requires *point_weights*.

keep_convex_hull
`bool` – Keep all tets within the convex hull. Default is false.

verbosity
`int` – Verbosity level. Ranges from 0 to 4:

0. no output
1. normal level of output
2. verbose output
3. more details
4. you must be debugging the tetgen code

vertices
`numpy.ndarray` – Vertices of the output tet mesh.

faces
`numpy.ndarray` – Faces of the output tet mesh.

voxels
`numpy.ndarray` – Voxels of the output tet mesh.

mesh
Mesh – Output tet mesh.

Example

```
>>> input_mesh = pymesh.generate_icosphere(1.0, [0.0, 0.0, 0.0]);
>>> tetgen = pymesh.tetgen();
>>> tetgen.points = input_mesh.vertices; # Input points.
>>> tetgen.triangles = input_mesh.faces; # Input triangles
>>> tetgen.max_tet_volume = 0.01;
>>> tetgen.verbosity = 0;
>>> tetgen.run(); # Execute tetgen
>>> mesh = tetgen.mesh; # Extract output tetrahedral mesh.
```

2.2.6 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.

While all solid geometry operations can be done as a sequence of binary boolean operations. It is beneficial sometimes to use *pymesh.CSGTree* for carrying out more complex operations.

class pymesh.**CSGTree**(*tree*)

Contructive Solid Geometry Tree.

Perhaps the best way of describing supported operations is using context free grammar:

- `mesh` operation: This operation is always a leaf node of the tree.

```
>>> tree = pymesh.CSGTree({"mesh": mesh});
```

- `union` operation:

```
>>> tree = pymesh.CSGTree({"union":
...           [TREE_1, TREE_2, ..., TREE_N]
...       });
```

- `intersection` operations:

```
>>> tree = pymesh.CSGTree({"intersection":
...           [TREE_1, TREE_2, ..., TREE_N]
...       });
```

- `difference` operations:

```
>>> tree = pymesh.CSGTree({"difference":
...           [TREE_1, TREE_2]
...      });
```

- `symmetric_difference` operations:

```
>>> tree = pymesh.CSGTree({"symmetric_difference":
...           [TREE_1, TREE_2]
...      });
```

Where `TREE_X` could be any of the nodes defined above.

A tree can be build up incrementally:

```
>>> left_tree = pymesh.CSGTree({"mesh": mesh_1});
>>> right_tree = pymesh.CSGTree({"mesh": mesh_2});
>>> tree = pymesh.CSGTree({"union": [left_tree, right_tree]});
>>> mesh = tree.mesh;
```

Or constructed from a dict:

```
>>> tree = pymesh.CSGTree({"union":
...           [{"mesh": mesh_1}, {"mesh": mesh_2}]
...      });
>>> mesh = tree.mesh
```

Convex hull

`pymesh.`**`convex_hull`**(*mesh*, *engine='auto'*, *with_timing=False*)

Compute the convex hull of an input mesh.

Parameters

- **`mesh`** (*Mesh*) – The input mesh.
- **`engine`** (`string`) – (optional) Convex hull engine name. Valid names are:
  - *auto*: Using the default engine.
  - *qhull*: Qhull convext hull library
  - ***cgal*: CGAL convex hull module (** 2D, 3D)
  - *triangle*: Triangle convex hull engine.
  - *tetgen*: Tetgen convex hull engine.
- **`with_timing`** (`boolean`) – (optional) Whether to time the code

Returns: The output mesh representing the convex hull. (and running time if *with_timing* is true.)

The following attributes are defined in the output mesh:

- "source_vertex": An array of source vertex indices into the input mesh.

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.

Mesh arrangement

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.

Subdivision

pymesh.**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.

Winding number query

pymesh.**compute_winding_number**(*mesh*, *queries*, *engine='auto'*)
Compute winding number with respect to *mesh* at *queries*.

Parameters

- **mesh** (*Mesh*) – The mesh for which winding number is evaluated.
- **queries** (numpy.ndarray) – N by 3 matrix of query points at which winding number is evaluated.
- **engine** (string) – (optional) Winding number computing engine name:
  - auto: use default engine (which is igl).
  - igl: use libigl's generalized winding number.
  - fast_winding_number: use code from fast winding number paper. It is faster than igl but can be less accurate sometimes.

Returns A list of size N, represent the winding numbers at each query points in order.

Slicing mesh

pymesh.**slice_mesh**(*mesh*, *direction*, *N*)
Slice a given 3D mesh N times along certain direciton.

Parameters

- **mesh** (*Mesh*) – The mesh to be sliced.
- **direction** (numpy.ndaray) – Direction orthogonal to the slices.
- **N** (*int*) – Number of slices.

Returns A list of *N* *Mesh* objects, each representing a single slice.

Distance to mesh query

pymesh.**distance_to_mesh**(*mesh*, *pts*, *engine='auto'*)
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.
- **engine** (string) – BVH engine name. Valid choices are "cgal", "geogram", "igl" if all dependencies are used. The default is "auto" where an available engine is automatically picked.

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.

2.2.7 Finite Element Matrix Assembly

In digital geometry processing, one often have to assemble matrices that corresponding to discrete differential operators. PyMesh provides a simple interface to assemble commonly used matrices.

class pymesh.**Assembler**(*mesh*, *material=None*)

Finite element matrix assembler

Example:

```
>>> mesh = pymesh.generate_icosphere(1.0, np.zeros(3), 3);
>>> assembler = pymesh.Assembler(mesh);
>>> L = assembler.assemble("laplacian");
>>> type(L)
<class 'scipy.sparse.csc.csc_matrix'>
```

This example assembles the Laplacian-Beltrami matrix used by many graphics applications. Other types of finite element matrices include:

- stiffness
- mass
- lumped_mass
- laplacian
- displacement_strain
- elasticity_tensor
- engineer_strain_stress
- rigid_motion
- gradient

2.2.8 Sparse Linear System Solver

Solving a sparse linear system is a common operation in geometry processing. In addition to the solvers provided by scipy, PyMesh brings the power of a number of state-of-the-art sparse solvers into python.

class pymesh.**SparseSolver**

Linear solver for solving sparse linear systems.

This class is a thin wrapper around sparse solvers supported by Eigen.

The following direct solvers are supported:

- `LLT`: Wrapper of Eigen::SimplicialLLT, SPD only.
- `LDLT`: Wrapper of Eigen::SimplicialLDLT, SPD only.
- `SparseLU`: Wrapper of Eigen::SparseLU.
- `SparseQR`: Wrapper of Eigen::SparseQR.
- `UmfPackLU`: Wrapper of Eigen::UmfPackLU. (Require SuiteSparse)
- `Cholmod`: Wrapper of Eigen::CholmodSupernodalLLT. (Require SuiteSparse)
- `PardisoLLT`: Wrapper of Eigen::PardisoLLT. SPD only. (Require Intel MKL)
- `PardisoLDLT`: Wrapper of Eigen::PardisoLDLT. SPD only. (Require Intel MKL)
- `PardisoLU`: Wrapper of Eigen::PardisoLU. (Require Intel MKL)

The following iterative solvers are supported:

- `CG`: Wrapper of Eigen::ConjugateGradient. SPD only.
- `LSCG`: Wrapper of Eigen::LeastSquaresConjugateGradient.
- `BiCG`: Wrapper of Eigen::BiCGSTAB.

supported_solvers
: `list` of `str` – The list of supported solvers.

tolerance
: `float` – The residual error threshold for stopping iterative solvers. Default is `Eigen::NumTraits<Float>::epsilon()`.

max_iterations
: `int` – The max iterations allowed for iterative solvers. Default is twice the number of columns of the matrix.

Example

For direct solvers:

```
>>> M = scipy.sparse.eye(100); # Build matrix.
>>> rhs = numpy.ones(100); # build right hand side.
>>> solver = pymesh.SparseSolver.create("LDLT");
>>> solver.compute(M);
>>> x = solver.solve(rhs);
```

For iterative solvers:

```
>>> M = scipy.sparse.eye(100); # Build matrix.
>>> rhs = numpy.ones(100); # build right hand side.
>>> solver = pymesh.SparseSolver.create("CG");
>>> solver.tolerance = 1e-10;
>>> solver.max_iterations = 50;
>>> solver.compute(M);
>>> x = solver.solve(rhs);
```

2.2.9 Miscellaneous functions

Matrix I/O

In addition to *numpy.save* and *numpy.load*, we also provide an alternative way of saving and loading *numpy.ndarray* to files.

pymesh.**save_matrix**(*filename*, *matrix*, *in_ascii=False*)
Save matrix into file in .dmat format.

Parameters

- **filename** (str) – Output file name.
- **matrix** (numpy.ndarray) – The matrix to save.
- **in_ascii** (*boolean*) – Whether to save matrix in ASCII. Default is false, which saves in binary format to save space.

pymesh.**load_matrix**(*filename*)
Load matrix from file (assuming .dmat format).

Mesh type conversion

pymesh.**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.**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.

Attribute type conversion

pymesh.**convert_to_vertex_attribute**(*mesh*, *attr*)
Convert attribute attr from either per-face or per-voxel attribute into per-vertex attribute.

Parameters

- **mesh** (*Mesh*) – Input mesh.
- **attr** (numpy.ndarray) – #vertices by k matrix of floats.

Returns Per-vertex attribute. The value at a vertex will be the average of the values at its neighboring faces or voxels.

pymesh.**convert_to_vertex_attribute_from_name**(*mesh*, *name*)
Same as *convert_to_vertex_attribute()* except looking up attribute values from the input mesh using name.

pymesh.**convert_to_face_attribute**(*mesh*, *attr*)
Convert attribute attr from either per-vertex or per-voxel attribute into per-face attribute.

Parameters

- **mesh** (*Mesh*) – Input mesh.
- **attr** (numpy.ndarray) – #faces by k matrix of floats.

Returns Per-face attribute. The value at a face will be the average of the values at its neighboring vertices or its neighboring voxels.

pymesh.**convert_to_face_attribute_from_name**(*mesh*, *name*)
Same as *convert_to_face_attribute()* except looking up attribute values from the input mesh using name.

pymesh.**convert_to_voxel_attribute**(*mesh*, *attr*)
Convert attribute attr from either per-vertex or per-face attribute into per-voxel attribute.

Parameters

- **mesh** (*Mesh*) – Input mesh.
- **attr** (numpy.ndarray) – #voxel by k matrix of floats.

Returns Per-voxel attribute. The value at a voxel will be the average of the values at its neighboring vertices or faces.

pymesh.**convert_to_voxel_attribute_from_name**(*mesh*, *name*)
Same as *convert_to_voxel_attribute()* except looking up attribute values from the input mesh using name.

Attribute mapping

pymesh.**map_vertex_attribute**(*mesh1*, *mesh2*, *attr_name*, *bvh=None*)
Map vertex attribute from mesh1 to mesh2 based on closest points.

Parameters

- **mesh1** (*Mesh*) – Source mesh, where the attribute is defined.
- **mesh2** (*Mesh*) – Target mesh, where the attribute is mapped to.
- **attr_name** (string) – Attribute name.
- **bvh** (BVH) – Pre-computed Bounded volume hierarchy if available.

A new attribute with name attr_name is added to mesh2.

pymesh.**map_face_attribute**(*mesh1*, *mesh2*, *attr_name*, *bvh=None*)
Map face attribute from mesh1 to mesh2 based on closest points.

Parameters

- **mesh1** (*Mesh*) – Source mesh, where the attribute is defined.
- **mesh2** (*Mesh*) – Target mesh, where the attribute is mapped to.

- **attr_name** (string) – Attribute name.
- **bvh** (BVH) – Pre-computed Bounded volume hierarchy if available.

A new attribute with name attr_name is added to mesh2.

pymesh.**map_corner_attribute**(*mesh1*, *mesh2*, *attr_name*, *bvh=None*)

Map per-vertex per-face attribute from mesh1 to mesh2 based on closest points.

Parameters

- **mesh1** (*Mesh*) – Source mesh, where the attribute is defined.
- **mesh2** (*Mesh*) – Target mesh, where the attribute is mapped to.
- **attr_name** (string) – Attribute name.
- **bvh** (BVH) – Pre-computed Bounded volume hierarchy if available.

A new attribute with name attr_name is added to mesh2.

Quaternion

class pymesh.**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[2].

z

float – same as quaternion[3].

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) – From 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.

to_matrix()
Convert to rotational matrix.

Returns The corresponding rotational matrix.

Return type numpy.ndarray

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

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

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

Returns The rotated vector.

Exact number types

pymesh.**Gmpz**
alias of mock.

pymesh.**Gmpq**
alias of mock.

Exact predicates

pymesh.**orient_2D**(*p1*, *p2*, *p3*)
Determine the orientation 2D points p1, p2, p3

Parameters **p1,p2,p3** – 2D points.

Returns positive if (p1, p2, p3) is in counterclockwise order. negative if (p1, p2, p3) is in clockwise order. 0.0 if they are collinear.

pymesh.**orient_3D**(*p1*, *p2*, *p3*, *p4*)
Determine the orientation 3D points p1, p2, p3, p4.

Parameters **p1,p2,p3,p4** – 3D points.

Returns positive if p4 is below the plane formed by (p1, p2, p3). negative if p4 is above the plane formed by (p1, p2, p3). 0.0 if they are coplanar.

pymesh.**in_circle**(*p1*, *p2*, *p3*, *p4*)
Determine if p4 is in the circle formed by p1, p2, p3.

Parameters **p1,p2,p3,p4** – 2D points. orient_2D(p1, p2, p3) must be postive, otherwise the result will be flipped.

Returns positive p4 is inside of the circle. negative p4 is outside of the circle. 0.0 if they are cocircular.

pymesh.**in_sphere**(*p1*, *p2*, *p3*, *p4*, *p5*)
Determine if p5 is in the sphere formed by p1, p2, p3, p4.

Parameters **p1,p2,p3,p4,p5** – 3D points. `orient_3D(p1, p2, p3, p4)` must be positive, otherwise the result will be flipped.

Returns positive p5 is inside of the sphere. negative p5 is outside of the sphere. 0.0 if they are cospherical.

The following predicates are built on top of above fundamental predicates.

`pymesh.`**`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 A boolean indicating whether `v0`, `v1` and `v2` are colinear.

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

`pymesh.`**`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 A `numpy.ndarray` of indices of degenerated faces.

`pymesh.`**`get_tet_orientations`**(*mesh*)
A thin wrapper of `get_tet_orientations_raw`.

`pymesh.`**`get_tet_orientations_raw`**(*vertices*, *tets*)
Compute orientation of each tet.

Parameters

- **vertice** (`numpy.ndarray`) – n by 3 matrix representing vertices.
- **tets** (`numpy.ndarray`) – m by 4 matrix of vertex indices representing tets.

Returns

A list of m floats, where

- Positive number => tet is positively oriented.
- 0 => tet is degenerate.
- Negative number => tet is inverted.

Mesh compression

`pymesh.`**`compress`**(*mesh*, *engine_name='draco'*)
Compress mesh data.

Parameters

- **mesh** (*Mesh*) – Input mesh.
- **engine_name** (`string`) – Valid engines are:

- draco: Google's Draco engine[1]

Returns A binary string representing the compressed mesh data.

A simple usage example:

```
>>> mesh = pymesh.generate_icosphere(1.0, [0, 0, 0])
>>> data = pymesh.compress(mesh)
>>> with open("out.drc", 'wb') as fout:
...     fout.write(data)
```

pymesh.**decompress**(*data*, *engine_name='draco'*)

Decompress mesh data.

Parameters

- **data** (string) – Binary string representing compressed mesh.
- **engine_name** (string) – Decompression engine name. Valid engines are:
  - draco: Google's Draco engine

Returns A mesh object encoded by the data.

A simple usage example:

```
>>> mesh = pymesh.generate_icosphere(1.0, [0, 0, 0])
>>> data = pymesh.compress(mesh)
>>> mesh2 = pymesh.decompress(data);
```

Mesh to graph

pymesh.**mesh_to_graph**(*mesh*)

Convert a mesh into a graph (V, E), where V represents the mesh vertices, and E represent the edges.

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

Returns The graph (V, E).

pymesh.**mesh_to_dual_graph**(*mesh*)

Convert a mesh into a dual graph (V, E), where V represents the mesh faces, and E represent the dual edges.

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

Returns The graph (V, E).

[1] Draco uses lossy compression. Both accuarcy and vertices/face order will be lost due to compression. Draco only works with triangular mesh or point cloud.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

A

add_attribute() (pymesh.Mesh method), 29
add_attribute() (pymesh.wires.WireNetwork method), 44
algorithm (pymesh.triangle attribute), 46
Assembler (class in pymesh), 54
attribute_names (pymesh.Mesh attribute), 29
attribute_names (pymesh.wires.WireNetwork attribute), 42
auto_hole_detection (pymesh.triangle attribute), 46

B

bbox (pymesh.Mesh attribute), 29
bbox (pymesh.wires.WireNetwork attribute), 42
bbox_center (pymesh.wires.WireNetwork attribute), 42
boolean() (in module pymesh), 49

C

center_at_origin() (pymesh.wires.WireNetwork method), 44
centroid (pymesh.wires.WireNetwork attribute), 42
coarsening (pymesh.tetgen attribute), 48
collapse_short_edges() (in module pymesh), 34
collapse_short_edges_raw() (in module pymesh), 34
compress() (in module pymesh), 60
compute_outer_hull() (in module pymesh), 51
compute_symmetry_orbits() (pymesh.wires.WireNetwork method), 44
compute_winding_number() (in module pymesh), 53
conforming_delaunay (pymesh.triangle attribute), 46
conjugate() (pymesh.Quaternion method), 59
convert_to_face_attribute() (in module pymesh), 57
convert_to_face_attribute_from_name() (in module pymesh), 57
convert_to_vertex_attribute() (in module pymesh), 56
convert_to_vertex_attribute_from_name() (in module pymesh), 57
convert_to_voxel_attribute() (in module pymesh), 57
convert_to_voxel_attribute_from_name() (in module pymesh), 57
convex_hull() (in module pymesh), 51
coplanar_tolerance (pymesh.tetgen attribute), 48
create_empty() (pymesh.wires.WireNetwork class method), 43
create_from_data() (pymesh.wires.WireNetwork class method), 43
create_from_file() (pymesh.wires.WireNetwork class method), 43
CSGTree (class in pymesh), 50

D

decompress() (in module pymesh), 61
detect_self_intersection() (in module pymesh), 38
dim (pymesh.Mesh attribute), 29
dim (pymesh.wires.WireNetwork attribute), 42
distance_to_mesh() (in module pymesh), 53

E

edges (pymesh.wires.WireNetwork attribute), 42
element_volumes (pymesh.Mesh attribute), 29
elements (pymesh.Mesh attribute), 29
exact_arithmetic (pymesh.tetgen attribute), 48
exact_arithmetic (pymesh.triangle attribute), 46

F

faces (pymesh.Mesh attribute), 28
faces (pymesh.tetgen attribute), 49
faces (pymesh.triangle attribute), 46
filter_edges() (pymesh.wires.WireNetwork method), 44
filter_vertices() (pymesh.wires.WireNetwork method), 44
form_mesh() (in module pymesh.meshio), 31
fromAxisAngle() (pymesh.Quaternion class method), 58
fromData() (pymesh.Quaternion class method), 58

G

generate_box_mesh() (in module pymesh), 40
generate_cylinder() (in module pymesh), 41
generate_dodecahedron() (in module pymesh), 42
generate_icosphere() (in module pymesh), 41

generate_regular_tetrahedron() (in module pymesh), 42
generate_tube() (in module pymesh), 41
get_attribute() (pymesh.Mesh method), 29
get_attribute() (pymesh.wires.WireNetwork method), 44
get_attribute_names() (pymesh.Mesh method), 30
get_attribute_names() (pymesh.wires.WireNetwork method), 44
get_degenerated_faces() (in module pymesh), 60
get_degenerated_faces_raw() (in module pymesh), 60
get_face_attribute() (pymesh.Mesh method), 30
get_tet_orientations() (in module pymesh), 60
get_tet_orientations_raw() (in module pymesh), 60
get_vertex_attribute() (pymesh.Mesh method), 29
get_vertex_neighbors() (pymesh.wires.WireNetwork method), 44
get_voxel_attribute() (pymesh.Mesh method), 30
Gmpq (in module pymesh), 59
Gmpz (in module pymesh), 59

H

has_attribute() (pymesh.Mesh method), 29
has_attribute() (pymesh.wires.WireNetwork method), 44
hex_to_tet() (in module pymesh), 56
holes (pymesh.triangle attribute), 45

I

in_circle() (in module pymesh), 59
in_sphere() (in module pymesh), 59
Inflator (class in pymesh.wires), 45
is_closed() (pymesh.Mesh method), 30
is_colinear() (in module pymesh), 60
is_edge_manifold() (pymesh.Mesh method), 30
is_manifold() (pymesh.Mesh method), 30
is_oriented() (pymesh.Mesh method), 30
is_vertex_attribute() (pymesh.wires.WireNetwork method), 44
is_vertex_manifold() (pymesh.Mesh method), 30

K

keep_convex_hull (pymesh.tetgen attribute), 49
keep_convex_hull (pymesh.triangle attribute), 45

L

load() (pymesh.wires.WireNetwork method), 43
load_from_file() (pymesh.wires.WireNetwork method), 43
load_from_raw() (pymesh.wires.WireNetwork method), 43
load_matrix() (in module pymesh), 56
load_mesh() (in module pymesh.meshio), 30

M

map_corner_attribute() (in module pymesh), 58
map_face_attribute() (in module pymesh), 57
map_vertex_attribute() (in module pymesh), 57
max_area (pymesh.triangle attribute), 45
max_areas (pymesh.triangle attribute), 45
max_iterations (pymesh.SparseSolver attribute), 55
max_num_steiner_points (pymesh.tetgen attribute), 48
max_num_steiner_points (pymesh.triangle attribute), 46
max_radius_edge_ratio (pymesh.tetgen attribute), 48
max_tet_volume (pymesh.tetgen attribute), 48
merge_coplanar (pymesh.tetgen attribute), 48
merge_meshes() (in module pymesh), 39
Mesh (class in pymesh), 28
mesh (pymesh.tetgen attribute), 49
mesh (pymesh.triangle attribute), 46
mesh_to_dual_graph() (in module pymesh), 61
mesh_to_graph() (in module pymesh), 61
min_angle (pymesh.triangle attribute), 45
min_dihedral_angle (pymesh.tetgen attribute), 48
minkowski_sum() (in module pymesh), 52

N

nodes (pymesh.Mesh attribute), 29
nodes_per_element (pymesh.Mesh attribute), 29
norm() (pymesh.Quaternion method), 59
normalize() (pymesh.Quaternion method), 59
num_components (pymesh.Mesh attribute), 29
num_edges (pymesh.wires.WireNetwork attribute), 42
num_elements (pymesh.Mesh attribute), 29
num_faces (pymesh.Mesh attribute), 29
num_nodes (pymesh.Mesh attribute), 29
num_surface_components (pymesh.Mesh attribute), 29
num_vertices (pymesh.Mesh attribute), 29
num_vertices (pymesh.wires.WireNetwork attribute), 42
num_volume_components (pymesh.Mesh attribute), 29
num_voxels (pymesh.Mesh attribute), 29

O

offset() (pymesh.wires.WireNetwork method), 44
optimization_level (pymesh.tetgen attribute), 48
orient_2D() (in module pymesh), 59
orient_3D() (in module pymesh), 59

P

Parameters (class in pymesh.wires), 45
partition_into_cells() (in module pymesh), 52
point_markers (pymesh.tetgen attribute), 48
point_weights (pymesh.tetgen attribute), 48
points (pymesh.tetgen attribute), 48
points (pymesh.triangle attribute), 45

Q

quad_to_tri() (in module pymesh), 56
Quaternion (class in pymesh), 58

R

regions (pymesh.triangle attribute), 46
remove_attribute() (pymesh.Mesh method), 30
remove_degenerated_triangles() (in module pymesh), 37
remove_degenerated_triangles_raw() (in module pymesh), 38
remove_duplicated_faces() (in module pymesh), 36
remove_duplicated_faces_raw() (in module pymesh), 36
remove_duplicated_vertices() (in module pymesh), 33
remove_duplicated_vertices_raw() (in module pymesh), 33
remove_isolated_vertices() (in module pymesh), 32
remove_isolated_vertices_raw() (in module pymesh), 32
remove_obtuse_triangles() (in module pymesh), 37
remove_obtuse_triangles_raw() (in module pymesh), 37
resolve_self_intersection() (in module pymesh), 38
rotate() (pymesh.Quaternion method), 59

S

save_matrix() (in module pymesh), 56
save_mesh() (in module pymesh.meshio), 30
save_mesh_raw() (in module pymesh.meshio), 31
scale() (pymesh.wires.WireNetwork method), 43
segments (pymesh.triangle attribute), 45
separate_mesh() (in module pymesh), 39
set_attribute() (pymesh.Mesh method), 30
set_attribute() (pymesh.wires.WireNetwork method), 44
set_profile() (pymesh.wires.Inflator method), 45
set_refinement() (pymesh.wires.Inflator method), 45
slice_mesh() (in module pymesh), 53
SparseSolver (class in pymesh), 54
split_boundary (pymesh.tetgen attribute), 48
split_boundary (pymesh.triangle attribute), 46
split_long_edges() (in module pymesh), 35
split_long_edges_raw() (in module pymesh), 35
subdivide() (in module pymesh), 53
submesh() (in module pymesh), 40
supported_solvers (pymesh.SparseSolver attribute), 55

T

tetgen (class in pymesh), 48
tetrahedralize() (in module pymesh), 47
tetrhaedra (pymesh.tetgen attribute), 48
Tiler (class in pymesh.wires), 45
to_matrix() (pymesh.Quaternion method), 59
tolerance (pymesh.SparseSolver attribute), 55
total_wire_length (pymesh.wires.WireNetwork attribute), 42
triangle (class in pymesh), 45
triangle_marker (pymesh.tetgen attribute), 48
triangles (pymesh.tetgen attribute), 48
triangles (pymesh.triangle attribute), 45
trim() (pymesh.wires.WireNetwork method), 44

V

verbosity (pymesh.tetgen attribute), 49
verbosity (pymesh.triangle attribute), 46
vertex_per_face (pymesh.Mesh attribute), 29
vertex_per_voxel (pymesh.Mesh attribute), 29
vertices (pymesh.Mesh attribute), 28
vertices (pymesh.tetgen attribute), 49
vertices (pymesh.triangle attribute), 46
vertices (pymesh.wires.WireNetwork attribute), 42
voronoi_edges (pymesh.triangle attribute), 46
voronoi_vertices (pymesh.triangle attribute), 46
voxels (pymesh.Mesh attribute), 28
voxels (pymesh.tetgen attribute), 49

W

w (pymesh.Quaternion attribute), 58
weighted_delaunay (pymesh.tetgen attribute), 49
wire_lengths (pymesh.wires.WireNetwork attribute), 42
WireNetwork (class in pymesh.wires), 42
write_to_file() (pymesh.wires.WireNetwork method), 43

X

x (pymesh.Quaternion attribute), 58

Y

y (pymesh.Quaternion attribute), 58

Z

z (pymesh.Quaternion attribute), 58