Intersector Documentation

Release 2.9

/ELSA/MU-10019/V2.9

Jun 07, 2019

CONTENTS

1 Preamble 1

2 List of functions 3

3 Contents 5
3.1 Main Functions 5
3.2 Transformation Functions 17
3.3 Extraction Functions 34
3.4 Check Functions 39
3.5 Conversion Functions 44

4 Index 45

CHAPTER

ONE

PREAMBLE

This module provides pre and post processing services relying on mesh-intersection computations on arbitrary polyhedral meshes.

It also gives auxiliary functions that transform topologically and geometrically polyhedral meshes which are useful in the process of mesh generation by intersection.

A mesh can be stored as an array (as defined in the Converter documentation) or in a zone node of a CGNS/python tree (pyTree).

This module is part of Cassiopee, a free open-source pre- and post-processor for CFD simulations.

For use with the array interface, you have to import Intersector module:

```
import Intersector as XOR
```

For use with the pyTree interface:

```
import Intersector.PyTree as XOR
```

CHAPTER

TWO

LIST OF FUNCTIONS

– Main Functions

Intersector.conformUnstr(a1[, a2, tol, ...])	Conformizes a1 (optionally with a2).
Intersector.booleanUnion(a1, a2[, tol, ...])	Computes the union between two closed-surface or two volume meshes.
Intersector.booleanIntersection(a1, a2[, ...])	Computes the intersection between two closed-surface or two volume meshes.
Intersector.booleanMinus(a1, a2[, tol, ...])	Computes the difference between two closed-surface or two volume meshes.
Intersector.diffSurf(a1, a2[, tol, ...])	Computes the difference between a volume mesh and a surface mesh.
Intersector.intersection(a1, a2[, tol])	Computes the intersection trace (a polyline) between two input closed surfaces.
Intersector.XcellN(coords, cellnfields, ...)	Computes the acurate cellN based on overlapped volumes.

– Collision predicates

Intersector.selfX(a)	Checks self-intersections in a mesh.
Intersector.getOverlappingFaces(a1, a2[, ...])	Returns the list of polygons in a1 and a2 that are overlapping.

– Transformation Functions

Intersector.triangulateBC(a, pgs)	Triangulates specified polygons of a volume mesh.
Intersector.triangulateExteriorFaces(a[, ...])	Triangulates exterior polygons of a volume mesh.

Continued on next page

Table 2.3 – continued from previous page

Intersector.reorientExternalFaces(a)	Reorients outward the external polygons of a mesh.
Intersector.convexifyFaces(a[, convexity_TOL])	Convexifies any non-convex polygon in a mesh.
Intersector.prepareCellsSplit(a[, PH_set, ...])	Splits some prescribed polygons following a prescribed splitting policy.
Intersector.splitNonStarCells(a[, ...])	Splits some non-centroid-star_shaped cells.
Intersector.simplifyCells(a, treat_externals)	Simplifies over-defined polyhedral cells (agglomerate some elligible polygons).
Intersector.agglomerateSmallCells(a[, vmin, ...])	Agglomerates prescribed cells.
Intersector.agglomerateNonStarCells(a)	Agglomerates non-centroid-star-shaped cells.
Intersector.agglomerateCellsWithSpecifiedFaces(a, pgs)	Agglomerates cells to make disappear specified polygons
Intersector.closeOctalCells(a)	Closes any polyhedral cell in an octree.
Intersector.adaptCells(a1, a2[, ...])	Adapts a polyhedral mesh a1 with repsect to a2 points.
Intersector.adaptBox(a[, box_ratio, itermax])	Adapts a bounding box to a cloud of interior points

– Extraction Functions

Intersector.extractPathologicalCells(a[, ...])	Extracts all cells that will probably cause trouble to a CFD solver.
Intersector.extractOuterLayers(a, N[, ...])	Extracts prescribed outer cell layers.

– Check Functions

Intersector.diffMesh(a1, a2)	Returns the difference between 2 meshes as 2 zones.
Intersector.checkCellsClosure(a)	Returns the first cell id that is non-closed.
Intersector.computeAspectRatio(a[, vmin])	Returns a field of aspect ratio.

– Conversion Functions

Intersector.convertNGON2DToNGON3D(a)	Converts a Cassiopee NGON Format for polygons (Face/Edge) to a Face/Node Format.

CHAPTER

THREE

CONTENTS

3.1 Main Functions

`Intersector.`**`conformUnstr`**(*a1*, *a2=None*, *tol=0.*, *left_or_right=0*, *itermax=10*)

Makes conformal a TRI or a BAR soup (i.e. a set of elements not necessarily connected as a mesh) by detecting and solving all the collisions between elements.

Colliding elements are cut to get a conformal set. Mixing types BAR and TRI is not currently handled.

Parameters

- **a1** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – First input mesh (BAR or TRI).
- **a2** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Second input mesh (BAR or TRI). If s2 is 'None' self-intersections are solved over s1.
- **tol** (`float`) – Merging tolerance when points (existing or computed by intersections) are too close.
- **left_or_right** (`0,1 or 2`) – Tells the function what to ouput : the transformed s1 (0), s2(1) or both (2).
- **itermax** (`int`) – Number of intersection/merging iterations. 10 is the default value.

Tips and Notes:

- Set itermax to 1. to improve speed and the Delaunay kernel robustness. The result might have poorer quality triangles though.
- Tolerance :
  - if tol > 0. : the value is used as an absolute overall tolerance
  - if tol = 0. : a value is computed as being 5% of the smallest edge length.

– if tol < 0. : MIN(5%, -tol) is used as a ratio to apply to the smallest edge length to get the tolerance.

Example of use:

- Makes conform a TRI or BAR soup (array):

```
# - conformUnstr (array) -
# Conforming 1 or 2 TRI/BAR together (same type for both operands
import Generator as G
import Intersector as XOR
import Converter as C
import Geom as D
from Geom.Parametrics import base
import Transform as T

s1 = D.sphere((0,0,0), 1, N=20)

s2 = D.surface(base['plane'], N=30)
s2 = T.translate(s2, (0.2,0.2,0.2))

s1 = C.convertArray2Tetra(s1); s1 = G.close(s1)
s2 = C.convertArray2Tetra(s2); s2 = G.close(s2)

x = XOR.conformUnstr(s1, s2, 0., 2)
C.convertArrays2File([x], 'out.plt')

c1 = D.circle((0,0,0), 1, N=100)
c2 = D.circle((0.2,0,0), 1, N=50)

c1 = C.convertArray2Tetra(c1); c1 = G.close(c1)
c2 = C.convertArray2Tetra(c2); c2 = G.close(c2)

x = XOR.conformUnstr(c1, c2, tol=0.)
C.convertArrays2File([x], 'out1.plt')
```

- Makes conform a TRI or BAR soup (pyTree):

```
# - conformUnstr (pyTree) -
# Conforming 1 or 2 TRI/BAR together (same type for both operands)
import Generator.PyTree as G
import Intersector.PyTree as XOR
import Converter.PyTree as C
import Geom.PyTree as D
from Geom.Parametrics import base
import Transform.PyTree as T
```

```
s1 = D.sphere((0,0,0), 1, N=20)

s2 = D.surface(base['plane'], N=30)
s2 = T.translate(s2, (0.2,0.2,0.2))

s1 = C.convertArray2Tetra(s1); s1 = G.close(s1)
s2 = C.convertArray2Tetra(s2); s2 = G.close(s2)

x = XOR.conformUnstr(s1, s2, tol=0.)
C.convertPyTree2File(x, 'out.plt')

c1 = D.circle((0,0,0), 1, N=100)
c2 = D.circle((0.2,0,0), 1, N=50)

c1 = C.convertArray2Tetra(c1); c1 = G.close(c1)
c2 = C.convertArray2Tetra(c2); c2 = G.close(c2)

x = XOR.conformUnstr(c1, c2, tol=0.)
C.convertPyTree2File(x, 'out1.plt')
```

`Intersector.`**booleanUnion**(*a1*, *a2*, *tol=0.*, *preserve_right=1*, *solid_right=1*, *agg_mode=1*, *extrude_pgs=[]*)

Creates a conformal union between two components, either TRI surfaces or Polyhedral volumes.

Parameters

- **a1** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – First mesh operand.
- **a2** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Second mesh operand.
- **tol** (`float`) – Merging tolerance when points (existing or computed by intersections) are too close.
- **preserve_right** (`0 or 1`) – Indicates the merging direction, either a1->a2 or a2->a1. If set to 1(0), it means a1->a2 (a2->a1), i.e. a2(a1)'s points are preserved.
- **solid_right** (`0 or 1`) – Indicates that the second operand is not penetrable, i.e. it is prioritized over the first operand a1.
- **agg_mode** (`0,1 or 2.`) – Option for agglomerating cut polygons : 0 to keep them as split triangles, 1 to get convex agglomerations and 2 to get a full agglomeration.
- **extrude_pgs** (`list of int`) – Optional list of polygons to extrude.

Prerequisites :

- External polygons must be oriented consistently and outwardly (use Transform.reorderAll before)

Tips and Notes:

- For assembling meshes, set solid_right to 1 and pass the prioritized mesh as second operand.
- extrude_pgs : required whenever a1 and a2 are in contact and a2 is prioritized : avoids to compute useless intersections by telling what are the indices of contact polygons in a2.

Example of use:

- Union of two spherical surface meshes (array):

```
# - booleanUnion (array) -
import Intersector as XOR
import Generator as G
import Converter as C
import Geom as D

s1 = D.sphere((0,0,0), 1, N=20)
s2 = D.sphere((0.,1.,0.), 1, N=30)

s1 = C.convertArray2Tetra(s1); s1 = G.close(s1)
s2 = C.convertArray2Tetra(s2); s2 = G.close(s2)

x = XOR.booleanUnion(s1, s2, tol=0.)
C.convertArrays2File([x], 'out.plt')
```

- Union of two volume meshes (pyTree):

```
# - boolean union (PyTree) -
import Intersector.PyTree as XOR
import Converter.PyTree as C

M1 = C.convertFile2PyTree('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1)

M2 = C.convertFile2PyTree('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2)

tol = -0.5e-3

x = XOR.booleanUnion(M1, M2, tol, preserve_right=1, solid_right=1)
```

```
t = C.newPyTree(['Base',2]); t[2][1][2].append(x)
C.convertPyTree2File(t, 'boolNGunion11.cgns')

x = XOR.booleanUnion(M1, M2, tol, preserve_right=0, solid_right=1)
t = C.newPyTree(['Base',2]); t[2][1][2].append(x)
C.convertPyTree2File(t, 'boolNGunion01.cgns')

x = XOR.booleanUnion(M1, M2, tol, preserve_right=1, solid_right=0)
t = C.newPyTree(['Base',2]); t[2][1][2].append(x)
C.convertPyTree2File(t, 'boolNGunion10.cgns')

x = XOR.booleanUnion(M1, M2, tol, preserve_right=0, solid_right=0)
t = C.newPyTree(['Base',2]); t[2][1][2].append(x)
C.convertPyTree2File(t, 'boolNGunion00.cgns')
```

`Intersector.`**`booleanIntersection`**(*a1, a2, tol=0., preserve_right=1, solid_right=1, agg_mode=1*)

Computes a conformal intersection between two components, either TRI surfaces or Polyhedral volumes.

Parameters

- **a1** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – First mesh operand.
- **a2** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Second mesh operand.
- **tol** (`float`) – Merging tolerance when points (existing or computed by intersections) are too close.
- **preserve_right** (`0 or 1`) – Indicates the merging direction, either a1->a2 or a2->a1. If set to 1(0), it means a1->a2 (a2->a1), i.e. a2(a1)'s points are preserved.
- **solid_right** (`0 or 1`) – Indicates that the second operand is not penetrable, i.e. it is prioritized over the first operand a1.
- **agg_mode** (`0,1 or 2.`) – Option for agglomerating cut polygons : 0 to keep them as split triangles, 1 to get convex agglomerations and 2 to get a full agglomeration.

Prerequisites :

- External polygons must be oriented consistently and outwardly (use Transform.reorderAll before)

Example of use:

- Intersection of two spherical surface meshes (array):

```
# - boolean intersection (array) -
import Intersector as XOR
import Generator as G
import Converter as C
import Geom as D

s1 = D.sphere((0,0,0), 1, N=20)
s2 = D.sphere((0.,1.,0.), 1, N=30)

s1 = C.convertArray2Tetra(s1); s1 = G.close(s1)
s2 = C.convertArray2Tetra(s2); s2 = G.close(s2)

x = XOR.booleanIntersection(s1, s2, tol=0.)
C.convertArrays2File([x], 'out.plt')
```

- Intersection of two volume meshes (pyTree):

```
# - boolean intersection (PyTree) -
import Intersector.PyTree as XOR
import Converter.PyTree as C

M1 = C.convertFile2PyTree('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1)

M2 = C.convertFile2PyTree('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2)

tol = 1.e-12
x = XOR.booleanIntersection(M1, M2, tol, preserve_right=1, solid_right=1)
t = C.newPyTree(['Base',2,x])
C.convertPyTree2File(t, 'boolNGinter11.cgns')

x = XOR.booleanIntersection(M1, M2, tol, preserve_right=0, solid_right=1)
C.convertPyTree2File(x, 'boolNGinter01.cgns')

x = XOR.booleanIntersection(M1, M2, tol, preserve_right=1, solid_right=0)
C.convertPyTree2File(x, 'boolNGinter10.cgns')

x = XOR.booleanIntersection(M1, M2, tol, preserve_right=0, solid_right=0)
C.convertPyTree2File(x, 'boolNGinter00.cgns')
```

`Intersector.`**`booleanMinus`**(*a1, a2, tol=0., preserve_right=1, solid_right=1, agg_mode=1*)

Computes a conformal difference between two components, either TRI surfaces or

Polyhedral volumes.

Parameters

- **a1** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – First mesh operand.
- **a2** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Second mesh operand.
- **tol** (float) – Merging tolerance when points (existing or computed by intersections) are too close.
- **preserve_right** (0 or 1) – Indicates the merging direction, either a1->a2 or a2->a1. If set to 1(0), it means a1->a2 (a2->a1), i.e. a2(a1)'s points are preserved.
- **solid_right** (0 or 1) – Indicates that the second operand is not penetrable, i.e. it is prioritized over the first operand a1.
- **agg_mode** (0,1 or 2.) – Option for agglomerating cut polygons : 0 to keep them as split triangles, 1 to get convex agglomerations and 2 to get a full agglomeration.

Prerequisites :

- External polygons must be oriented consistently and outwardly (use Transform.reorderAll before)

Example of use:

- Difference of two spherical surface meshes (array):

```
# - booleanMinus (array) -
import Intersector as XOR
import Generator as G
import Converter as C
import Geom as D

s1 = D.sphere((0,0,0), 1, N=20)
s2 = D.sphere((0.,1.,0.), 1, N=30)

s1 = C.convertArray2Tetra(s1); s1 = G.close(s1)
s2 = C.convertArray2Tetra(s2); s2 = G.close(s2)

x = XOR.booleanMinus(s1, s2, tol=0.)
C.convertArrays2File([x], 'out.plt')
```

- Difference of two volume meshes (pyTree):

```
# - boolean minus (PyTree) -
import Intersector.PyTree as XOR
import Converter.PyTree as C

M1 = C.convertFile2PyTree('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1)

M2 = C.convertFile2PyTree('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2)

tol = 1.e-12

x = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1)
t = C.newPyTree(['Base',2]); t[2][1][2].append(x)
C.convertPyTree2File(t, 'boolNGminus11.cgns')

x = XOR.booleanMinus(M1, M2, tol, preserve_right=0, solid_right=1)
t = C.newPyTree(['Base',2]); t[2][1][2].append(x)
C.convertPyTree2File(t, 'boolNGminus01.cgns')

x = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=0)
t = C.newPyTree(['Base',2]); t[2][1][2].append(x)
C.convertPyTree2File(t, 'boolNGminus10.cgns')

x = XOR.booleanMinus(M1, M2, tol, preserve_right=0, solid_right=0)
t = C.newPyTree(['Base',2]); t[2][1][2].append(x)
C.convertPyTree2File(t, 'boolNGminus00.cgns')
```

`Intersector.`**`intersection`**(*a1*, *a2*, *tol=0.*)

Returns the 'BAR' contour defining the intersection between two TRI-surfaces.

Parameters

- **a1** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – First mesh operand.
- **a2** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Second mesh operand.
- **tol** (`float`) – Merging tolerance when points (existing or computed by intersections) are too close.

Example of use:

- Circular trace of the intersection between two spheres (array):

```
# - intersection (array) -
import Intersector as XOR
```

```
import Generator as G
import Converter as C
import Geom as D

s1 = D.sphere((0,0,0), 1, N=20)
s2 = D.sphere((0.,1.,0.), 1, N=30)
s1 = C.convertArray2Tetra(s1); s1 = G.close(s1)
s2 = C.convertArray2Tetra(s2); s2 = G.close(s2)
x = XOR.intersection(s1, s2, tol=0.)
C.convertArrays2File([x], 'out.plt')
```

- Circular trace of the intersection between two spheres (pyTree):

```
# - intersection (pyTree) -
import Intersector.PyTree as XOR
import Generator.PyTree as G
import Converter.PyTree as C
import Geom.PyTree as D

s1 = D.sphere((0,0,0), 1, N=20)
s2 = D.sphere((0.,1.,0.), 1, N=30)
s1 = C.convertArray2Tetra(s1); s1 = G.close(s1)
s2 = C.convertArray2Tetra(s2); s2 = G.close(s2)
x = XOR.intersection(s1, s2, tol=0.)
C.convertPyTree2File(x, 'out.cgns')
```

`Intersector.`**`diffSurf`**(*a1, a2, tol=0., preserve_right=1, agg_mode=1*)

Cut-cell function : Computes a conformal difference between a volume mesh and a surface mesh

Parameters

- **a1** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – First mesh operand.
- **a2** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Second mesh operand.
- **tol** (`float`) – Merging tolerance when points (existing or computed by intersections) are too close.
- **preserve_right** (`0 or 1`) – Indicates the merging direction, either a1->a2 or a2->a1. If set to 1(0), it means a1->a2 (a2->a1), i.e. a2(a1)'s points are preserved.
- **solid_right** (`0 or 1`) – Indicates that the second operand is not penetrable, i.e. it is prioritized over the first operand a1.

- **agg_mode** (0,1 or 2.) – Option for agglomerating cut polygons : 0 to keep them as split triangles, 1 to get convex agglomerations and 2 to get a full agglomeration.

Prerequisites :

- External polygons must be oriented consistently and outwardly (use Transform.reorderAll before)
- The surface format must be an NGON Face/Node (apply before Intersector.convertNGON2DToNGON3D on the surface)

Example of use:

- Cut-cell mesh with an octree and a sphere (array):

```
# - boolean diffSurf (array) -
import Generator as G
import Geom as D
import Converter as C
import Intersector as XOR

# octree
s = D.sphere((0,0,0), 1., 100); snear = 0.1
t = G.octree([s],[snear], dfar=5., balancing=1,ratio=2)

# ngon conversion
t = C.convertArray2NGon(t)
# ngon conformization
t = C.conformizeNGon(t); t = G.close(t)
# ngon close cells
t = XOR.closeOctalCells(t)
#t = XOR.reorientExternalFaces(t)

# ngon conversion of the sphere
s = C.convertArray2NGon(s)
# ngon conversion to the nuga format
s = XOR.convertNGON2DToNGON3D(s)
#s = XOR.reorientExternalFaces(s)

# Boolean operation
x = XOR.diffSurf(t, s, tol = 0., preserve_right=1, agg_mode=2) # agg_mode=2 :␣
↪full mode aggregation

C.convertArrays2File([x], 'out.plt')
```

- Cut-cell mesh with an octree and a sphere (pyTree):

```
# - boolean diffSurf (PyTree) -
import Generator.PyTree as G
import Geom.PyTree as D
import Converter.PyTree as C
import Intersector.PyTree as XOR

# octree
s = D.sphere((0,0,0), 1., 100); snear = 0.1
t = G.octree([s],[snear], dfar=5., balancing=1,ratio=2)

# ngon conversion
t = C.convertArray2NGon(t)
# ngon conformization
t = C.conformizeNGon(t); t = G.close(t)
# ngon close cells
t = XOR.closeOctalCells(t)
#t = XOR.reorientExternalFaces(t)

# ngon conversion of the sphere
s = C.convertArray2NGon(s)
# ngon conversion to the nuga format
s = XOR.convertNGON2DToNGON3D(s)
#s = XOR.reorientExternalFaces(s)

# Boolean operation
x = XOR.diffSurf(t, s, tol = 0., preserve_right=1, agg_mode=2) # agg_mode=2 :␣
↪full mode aggregation

t = C.newPyTree(['Base',2]); t[2][1][2].append(x)
C.convertPyTree2File(t, 'diffs.cgns')
```

Intersector.**XcellN**(*a*, *cellnfields*, *maskingMesh*, *wall_pgl=[]*, *ghost_pgl=[]*)

Computes the cell nature field of a background mesh (a) in an overset configuration : similarly to the blankCells functions, the input maskingMesh are volume meshes that hide a.

The computed celln is accurate, giving a floating value ranging from 0 (fully masked) to 1 (fully visible).

The input grids (a and makingMesh) are defined by coordinates located at nodes as a list of arrays.

Parameters

- **a** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Mesh where to compute XcellN
- **cellnfields** ([array, list of arrays]) – celln array for a
- **maskingMesh** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Prioritized mesh that hides a
- **wall_pgl** (list of int) – Optional list of polygons to treat as walls.
- **ghost_pgl** (list of int) – Optional list of polygons to extrude.

Tips and Notes:

- Warning: location of celln must be located at centers.
- Warning: In order to set the celln to 0. inside blanking bodies, you need to create BCWall type boundaries on the body faces.

Example of use:

- Computing an accurate cellN (array):

```
# - XcellN (array) -
import Converter as C
import Generator as G
import Intersector as XOR

# Test 1
# Mask
masking = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
masking = C.convertArray2NGon(masking)

# Mesh to blank
a = G.cart((-3.,-3.,-3.), (0.5,0.5,0.5), (20,20,20))
a = C.convertArray2NGon(a)

# celln init
ca = C.node2Center(a)
ca = C.initVars(ca, 'cellN', 1.)
cn = C.extractVars(ca, ['cellN'])

# Blanking
celln = XOR.XcellN([a], [cn], masking)

celln = C.center2Node(celln[0])
a = C.addVars([a, celln])

C.convertArrays2File(a, 'out.plt')
```

- Computing an accurate cellN (pyTree):

```
# - XcellN (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Intersector.PyTree as XOR
import Geom.PyTree as D

# Test 1
# Mask
masking = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
masking = C.convertArray2NGon(masking)
# Mesh to blank
bgm = G.cart((-3.,-3.,-3.), (0.5,0.5,0.5), (20,20,20))
t = C.newPyTree(['Cart', bgm])
t = C.convertArray2NGon(t)

# celln init
C._initVars(t, 'centers:cellN', 1.)
# Blanking with floating cellN computation
t = XOR.XcellN(t, [[masking]], [])
C.convertPyTree2File(t, 'out1.cgns')

# Test 2
# Tet mask
masking = D.sphere((0,0,0), 15., 30)
masking = C.convertArray2Tetra(masking)
masking = G.close(masking)
masking = G.tetraMesher(masking, algo=1)
#C.convertPyTree2File(masking, 'sph.cgns')
# Mesh to blank
bgm = G.cart((-5.,-5.,-5.), (0.8,0.8,0.8), (40,40,40))
t = C.newPyTree(['Cart', bgm])
t = C.convertArray2NGon(t)
# celln init
C._initVars(t, 'centers:cellN', 1.)
# Blanking
t = XOR.XcellN(t, [[masking]], [])
C.convertPyTree2File(t, 'out2.cgns')
```

3.2 Transformation Functions

Intersector.**triangulateBC**(*a*, *bctype*)

Triangulates the prescribed BC type polygons of a volume mesh.

Parameters

- **a** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Input mesh
- **bctype** (string) – boundary type ('BCWall', . . .).

Example of use:

- BC polygons triangulation (pyTree):

```
# - triangulateExteriorFaces (PyTree) -
import Intersector.PyTree as XOR
import Converter.PyTree as C

t = C.convertFile2PyTree('boolNG_M1.tp')

t = C.convertArray2NGon(t)

t = C.fillEmptyBCWith(t, 'wall', 'BCWall', dim=3)

XOR.triangulateBC(t, 'BCWall')

C.convertPyTree2File(t, 'out.cgns')
```

Intersector.**triangulateExteriorFaces**(*a*, *in_or_out=2*)

Triangulates the prescribed external polygons of a volume mesh.

Parameters

- **a** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Input mesh
- **in_or_out** (0,1 or 2) – In case of a non-connex mesh (i.e. whith holes like an external airflow mesh around bodies), set to 0 for processing only body walls, set to 1 for processing only the outer boundary, or 2 for processing all of them.

Example of use:

- External polygons triangulation (array):

```
# - triangulateExteriorFaces (array) -
import Intersector as XOR
import Converter as C

m = C.convertFile2Arrays('boolNG_M1.tp')
m = C.convertArray2NGon(m[0])

m = XOR.triangulateExteriorFaces(m)
```

```
C.convertArrays2File([m], 'out.plt')
```

- External polygons triangulation (pyTree):

```
# - triangulateExteriorFaces (PyTree) -
import Intersector.PyTree as XOR
import Converter.PyTree as C

t = C.convertFile2PyTree('boolNG_M1.tp')
t = C.convertArray2NGon(t)

t = XOR.triangulateExteriorFaces(t)
C.convertPyTree2File(t, 'out.cgns')
```

Intersector.**reorientExternalFaces**(*a*)

Reorients outward the external polygons of a mesh.

Example of use:

- Reorientation (array):

```
# - boolean reorientExternalFaces (array) -
import Generator as G
import Converter as C
import Intersector as XOR


a = G.cartHexa((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
a = C.convertArray2NGon(a)
a = XOR.reorientExternalFaces(a)

C.convertArrays2File([a], 'out.plt')
```

- Reorientation (pyTree):

```
# - boolean reorientExternalFaces (array) -
import Generator.PyTree as G
import Converter.PyTree as C
import Intersector.PyTree as XOR


a = G.cartHexa((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
a = C.convertArray2NGon(a)
```

```
a = XOR.reorientExternalFaces(a)

C.convertPyTree2File(a, 'out.cgns')
```

`Intersector.`**`convexifyFaces`**(*a*, *convexity_TOL = 1.e-8*)

Makes a convex decomposition of any concave polygon in a mesh.

Parameters

- **a** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Input mesh
- **convexity_TOL** (`float`) – convexity angle threshold

Example of use:

- Convexify polygons (array):

```
# - convexifyFaces (array) -
# convexify any concave polygon in the mesh
import Intersector as XOR
import Converter as C

M1 = C.convertFile2Arrays('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1[0])

M2 = C.convertFile2Arrays('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2[0])

tol = -0.5e-3

m = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=2)
↪#full agg to convexify afterward
#C.convertArrays2File([m], 'i.plt')
m = XOR.convexifyFaces(m)

C.convertArrays2File([m], 'out.plt')
```

- Convexify polygons (pyTree):

```
# - convexifyFaces (pyTree) -
# convexify any concave polygon in the mesh
import Intersector.PyTree as XOR
import Converter.PyTree as C
```

```
M1 = C.convertFile2PyTree('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1)

M2 = C.convertFile2PyTree('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2)

tol = -0.5e-3

m = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=2)
↪#full agg to convexify afterward
m = XOR.convexifyFaces(m)

C.convertPyTree2File(m, 'out.cgns')
```

`Intersector.`**`prepareCellsSplit`**(*a*, *PH_set = 1*, *split_policy = 0*, *PH_conc_threshold = 1./3.*, *PH_cvx_threshold = 0.05*, *PG_cvx_threshold = 1.e-8*)

Prepares the bad cells split (`splitNonStarCells`) by splitting some of their polygons with a prescribed policy : convexification, starification.

Parameters

- **a** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Input mesh
- **PH_set** (`0 or 1`) – PH to process. 0 for concave cells or 1 for non-centroid-star_shaped cells
- **split_policy** (`0,1 or 2`) – 0 : convexify concave pgs. 1 : starify concave pgs from worst vertex. 2 : starify concave pgs from concave-chains ends.
- **PH_conc_threshold** (`float`) – Concavity dihedral angle threshold for cells
- **PH_cvx_threshold** (`float`) – Convexity dihedral angle threshold for cells
- **PG_cvx_threshold** (`float`) – Convexity angle threshold for polygons

Example of use:

- prepareCellsSplit (array):

```
# - convexify any concave polygon in the mesh (array) -
import Intersector as XOR
import Converter as C
```

```
M1 = C.convertFile2Arrays('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1[0])

M2 = C.convertFile2Arrays('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2[0])

tol = -0.5e-3

m = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=2)
↪#full agg to convexify afterward
#C.convertArrays2File([m], 'i.plt')
m = XOR.prepareCellsSplit(m, PH_set = 0, split_policy = 2, PH_conc_threshold =␣
↪1./3., PH_cvx_threshold = 0.05, PG_cvx_threshold = 1.e-2)

C.convertArrays2File([m], 'out.plt')
```

- prepareCellsSplit (pyTree):

```
# - convexify any concave polygon in the mesh (array) -
import Intersector.PyTree as XOR
import Converter.PyTree as C

M1 = C.convertFile2PyTree('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1)

M2 = C.convertFile2PyTree('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2)

tol = -0.5e-3

m = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=2)
↪#full agg to convexify afterward
#C.convertArrays2File([m], 'i.plt')
m = XOR.prepareCellsSplit(m, PH_set = 0, split_policy = 2, PH_conc_threshold =␣
↪1./3., PH_cvx_threshold = 0.05, PG_cvx_threshold = 1.e-2)

C.convertPyTree2File(m, 'out.cgns')
```

`Intersector.`**`splitNonStarCells`**(*a*, *PH_conc_threshold = 1./3.*, *PH_cvx_threshold = 0.05*, *PG_cvx_threshold = 1.e-8*)

First strategy to eradicate bad cells : Splits non-centroid-star-shaped (NCSS) cells into two cells. These cells might be NCSS as well so this function should be called several times to get rid off the pathologies. Some call agglomerateSmallCells should be done afterwards to balance the aspect ratio.

Parameters

- **a** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Input mesh
- **PH_conc_threshold** (float) – Concavity dihedral angle threshold for cells
- **PH_cvx_threshold** (float) – Convexity dihedral angle threshold for cells
- **PG_cvx_threshold** (float) – Convexity angle threshold for polygons

Tips and Notes:

- Call prepareCellsSplit before this function to ensure to process as much pathologies as possible.

Example of use:

- splitNonStarCells (array):

```
# - convexify any concave polygon in the mesh (array) -
import Intersector as XOR
import Converter as C

M1 = C.convertFile2Arrays('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1[0])

M2 = C.convertFile2Arrays('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2[0])

tol = -0.5e-3

m = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=1)

m = XOR.simplifyCells(m, 1)
m = XOR.prepareCellsSplit(m, PH_set = 0, split_policy = 0, PH_conc_threshold =␣
↪1./3., PH_cvx_threshold = 0.05, PG_cvx_threshold = 1.e-8)
m= XOR.splitNonStarCells(m, PH_conc_threshold = 1./3., PH_cvx_threshold = 0.05,␣
↪PG_cvx_threshold = 1.e-8)

C.convertArrays2File([m], 'out.plt')
```

- splitNonStarCells (pyTree):

```
# - convexify any concave polygon in the mesh (array) -
import Intersector.PyTree as XOR
import Converter.PyTree as C

M1 = C.convertFile2PyTree('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1)

M2 = C.convertFile2PyTree('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2)

tol = -0.5e-3

m = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=1)

m = XOR.simplifyCells(m, 1)
m = XOR.prepareCellsSplit(m, PH_set = 0, split_policy = 0, PH_conc_threshold =␣
↪1./3., PH_cvx_threshold = 0.05, PG_cvx_threshold = 1.e-8)
m= XOR.splitNonStarCells(m, PH_conc_threshold = 1./3., PH_cvx_threshold = 0.05,␣
↪PG_cvx_threshold = 1.e-8)

C.convertPyTree2File(m, 'out.cgns')
```

`Intersector.`**`simplifyCells`**(*a*, *treat_externals*, *angular_threshold = 1.e-12*)

Agglomerates superfluous polygons that over-defines cells. After agglomerating (e.g. after calling `agglomerateSmallCells`) , we end up with cells that are multiply-connected, i.e. they share more than one polygon. If 2 cells share 2 polygons that are connected (sharing an edge) and their dihedral angle is below the angular_threshold, then the 2 polygon are agglomerated upon exit.

Parameters

- **a** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Input mesh
- **treat_externals** (`0 or 1`) – Process outer polygons (1) or not (0).
- **angular_threshold** (`float`) – Largest angular deviation admitted between polygons in order to allow the agglomeration for them.

Example of use:

- simplifyCells (array):

```
# - convexify any concave polygon in the mesh (array) -
import Intersector as XOR
import Converter as C
```

```
M1 = C.convertFile2Arrays('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1[0])

M2 = C.convertFile2Arrays('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2[0])

tol = -0.5e-3

m = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=1)
#C.convertArrays2File([m], 'i.plt')

m = XOR.simplifyCells(m, 1)

C.convertArrays2File([m], 'out.plt')
```

- simplifyCells (pyTree):

```
# - convexify any concave polygon in the mesh (array) -
import Intersector.PyTree as XOR
import Converter.PyTree as C

M1 = C.convertFile2PyTree('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1)

M2 = C.convertFile2PyTree('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2)

tol = -0.5e-3

m = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=1)

m = XOR.simplifyCells(m, 1)

C.convertPyTree2File(m, 'out.cgns')
```

`Intersector.`**`agglomerateSmallCells`**(*a*, *vmin=0.*, *vratio=1000.*)

Agglomerates cells that are too small (below vmin) or having a poor aspect ratio with a neighbor (below vratio) with the best neighbor available. The agglomeration process does not create non-star-shaped agglomerates.

Parameters

- **a** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Input mesh

- **vmin** (float) – volume threshold.
- **vratio** (float) – aspect ratio threshold.

Tips and Notes:

- See computeAspectRatio to get the definition of the computed aspect ratio.

Example of use:

- agglomerateSmallCells (array):

```
# - boolean diffSurf (array) -
import Generator as G
import Geom as D
import Converter as C
import Intersector as XOR

# octree
s = D.sphere((0,0,0), 1., 100); snear = 0.1
t = G.octree([s], [snear], dfar=5., balancing=1, ratio=2)

# ngon conversion
t = C.convertArray2NGon(t)
# ngon conformization
t = C.conformizeNGon(t); t = G.close(t)
# ngon close cells
t = XOR.closeOctalCells(t)
#t = XOR.reorientExternalFaces(t)

# ngon conversion of the sphere
s = C.convertArray2NGon(s)
# ngon conversion to the nuga format
s = XOR.convertNGON2DToNGON3D(s)
#s = XOR.reorientExternalFaces(s)

# Boolean operation
x = XOR.diffSurf(t, s, tol = 0., preserve_right=1, agg_mode=2) # agg_mode=2 :␣
↪full mode aggregation
C.convertArrays2File(x, 'diffsurf.plt')
x = XOR.agglomerateSmallCells(x, 0., 10.)
C.convertArrays2File(x, 'agg.plt')
```

- agglomerateSmallCells (pyTree):

```
# - boolean diffSurf (PyTree) -
import Generator.PyTree as G
```

```
import Geom.PyTree as D
import Converter.PyTree as C
import Intersector.PyTree as XOR

# octree
s = D.sphere((0,0,0), 1., 100); snear = 0.1
t = G.octree([s],[snear], dfar=5., balancing=1,ratio=2)

# ngon conversion
t = C.convertArray2NGon(t)
# ngon conformization
t = C.conformizeNGon(t); t = G.close(t)
# ngon close cells
t = XOR.closeOctalCells(t)
#t = XOR.reorientExternalFaces(t)

# ngon conversion of the sphere
s = C.convertArray2NGon(s)
# ngon conversion to the nuga format
s = XOR.convertNGON2DToNGON3D(s)
#s = XOR.reorientExternalFaces(s)

# Boolean operation
x = XOR.diffSurf(t, s, tol = 0., preserve_right=1, agg_mode=2) # agg_mode=2 :␣
↪full mode aggregation

x = XOR.agglomerateSmallCells(x, 0., 10.)

t = C.newPyTree(['Base',2]); t[2][1][2].append(x)
C.convertPyTree2File(t, 'diffs.cgns')
```

Intersector.**agglomerateNonStarCells**(*a*)

Agglomerate cells that are non-centroid-star-shaped. The agglomeration process does not create non-star-shaped agglomerates.

Parameters a (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Input mesh

Example of use:

- agglomerateNonStarCells (array):

```
# - convexify any concave polygon in the mesh (array) -
import Intersector as XOR
import Converter as C

M1 = C.convertFile2Arrays('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1[0])

M2 = C.convertFile2Arrays('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2[0])

tol = -0.5e-3

m = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=1)
#C.convertArrays2File([m], 'i.plt')

m = XOR.agglomerateNonStarCells(m)

C.convertArrays2File(m, 'out.plt')
```

- agglomerateNonStarCells (pyTree):

```
# - convexify any concave polygon in the mesh (array) -
import Intersector.PyTree as XOR
import Converter.PyTree as C

M1 = C.convertFile2PyTree('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1)

M2 = C.convertFile2PyTree('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2)

tol = -0.5e-3

m = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=1)

m = XOR.agglomerateNonStarCells(m)

C.convertPyTree2File(m, 'out.cgns')
```

Intersector.**agglomerateCellsWithSpecifiedFaces**(*a*, *pgs*, *simplify*)

Agglomerate cells that are non-centroid-star-shaped. The agglomeration process does not create non-star-shaped agglomerates.

Parameters

- **a** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Input mesh
- **pgs** (`list of integers`) – list of polygons to remove

Example of use:

- agglomerateCellsWithSpecifiedFaces (array):

```
import Generator as G
import Transform as T
import Converter as C
import Converter as I
import Intersector as XOR
import KCore.test as test
import Post as P

t1 = G.cart((0,0,0), (1,1,1), (10,10,10))
t1 = C.convertArray2NGon(t1); t1 = G.close(t1)
t2 = G.cart((1.,0,0), (1,1,1), (10,10,10))
t2 = C.convertArray2NGon(t2); t2 = G.close(t2)

# test 1 : volume/volume
res = XOR.getOverlappingFaces(t1, t2, RTOL = 0.05, ps_min = 0.95)

# create a list of polygon list (t1), one list per zone

t = XOR.agglomerateCellsWithSpecifiedFaces(t1, res[0])

C.convertArrays2File([t], "out.plt")

#test 2 : volume/surface

t2 = P.exteriorFaces(t2)
t2 = XOR.convertNGON2DToNGON3D(t2)

res = XOR.getOverlappingFaces(t1, t2, RTOL = 0.05, ps_min = 0.95)

t = XOR.agglomerateCellsWithSpecifiedFaces(t1, res[0])

C.convertArrays2File([t], "out1.plt")
```

- agglomerateCellsWithSpecifiedFaces (pyTree):

```
import Generator.PyTree as G
import Transform.PyTree as T
```

```
import Converter.PyTree as C
import Converter.Internal as I
import Intersector.PyTree as XOR
import KCore.test as test
import Post.PyTree as P

t1 = G.cart((0,0,0), (1,1,1), (10,10,10))
t1 = C.convertArray2NGon(t1); t1 = G.close(t1)
t2 = G.cart((1.,0,0), (1,1,1), (10,10,10))
t2 = C.convertArray2NGon(t2); t2 = G.close(t2)

# test 1 : volume/volume
res = XOR.getOverlappingFaces(t1, t2, RTOL = 0.05, ps_min = 0.95)

# create a list of polygon list (t1), one list per zone
nb_zones = len(res)
t1zones_pgids = []
for i in range(nb_zones):
  t1zones_pgids.append(res[i][0])

t = XOR.agglomerateCellsWithSpecifiedFaces(t1, t1zones_pgids)

C.convertPyTree2File(t, "out.cgns")

#test 2 : volume/surface

t2 = P.exteriorFaces(t2)
t2 = XOR.convertNGON2DToNGON3D(t2)

res = XOR.getOverlappingFaces(t1, t2, RTOL = 0.05, ps_min = 0.95)

t1zones_pgids = []
for i in range(nb_zones):
  t1zones_pgids.append(res[i][0])

t = XOR.agglomerateCellsWithSpecifiedFaces(t1, t1zones_pgids)

C.convertPyTree2File(t, "out1.cgns")
```

Tips and Notes:

- When asembling 2 meshes m1 and m2 where m2 is priorized , to improve the assembly quality, do before calling the boolean union:

1. getOverlappingFaces (m1, skin(m2)) where skin(m2) is the external polygonal skin of m2

2. agglomerateCellsWithSpecifiedFaces on m1 with the above list of polygons

---

Intersector.**closeOctalCells**(*a*)

Closes any polyhedral cell in a 2:1 octree.

Parameters **a** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Input mesh

Example of use:

- closeOctalCells (array):

```
# - triangulateExteriorFaces (array) -
import Intersector as XOR
import Converter as C

m = C.convertFile2Arrays('boolNG_M1.tp')
m = C.convertArray2NGon(m[0])

m = XOR.closeOctalCells(m)
C.convertArrays2File([m], 'out.plt')
```

- closeOctalCells (pyTree):

```
# - triangulateExteriorFaces (array) -
import Intersector.PyTree as XOR
import Converter.PyTree as C
import KCore.test as test

m = C.convertFile2PyTree('boolNG_M1.tp')
m = C.convertArray2NGon(m)

m = XOR.closeOctalCells(m)
C.convertPyTree2File(m, 'out.cgns')
```

Tips and Notes:

- Do this transformation whenever you need to use a surface algorithm on the octree (e.g. `reorientExternalFaces`)

Intersector.**adaptCells**(*a1*, *a2*, *sensor_type=0*)

Adapts a1 cells (only TETRA and HEXA cells currently) with respect to a2 points. Adaptation is a per-cell octal 2:1 decomposition. With a sensor_type equal to 0, a2 points are only considered : a1 will be refined such any a1 cell contains at most 1 a2's point. With a sensor_type equal to 1, a2's connectivity is also taken into account by adding refinement wherever a1 cells are crossed by a2 edges.

Parameters

- **a1** (`[array] or [ single zone pyTree (currently)]`) – Input mesh (NGON format)
- **a2** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Source points or source mesh
- **sensor_type** (`int`) – type of sensor. Using only the point cloud (0) or both points and connectivity via instersections (1)

Example of use:

- adaptCells (array):

```
# - adapts a cells with respect to b points (array) -
import Intersector as XOR
import Converter as C
import Generator as G

a = G.cartHexa((0.,0.,0.), (0.1,0.1,0.1), (5,5,5))
a = C.convertArray2NGon(a); a = G.close(a)
#C.convertArrays2File([a], 'a.plt')
b = G.cartHexa((0.,0.,0.), (0.005,0.005,0.005), (5,5,5))
#C.convertArrays2File([b], 'b.plt')

m = XOR.adaptCells(a,b, sensor_type=0)

m = XOR.closeOctalCells(m)
C.convertArrays2File([m], 'out.plt')

m = XOR.adaptCells(a,b, sensor_type=1)

m = XOR.closeOctalCells(m)
C.convertArrays2File([m], 'xout.plt')
```

- adaptCells (pyTree):

```
# - adapts a cells with respect to b points (PyTree) -
import Intersector.PyTree as XOR
import Converter.PyTree as C
import Generator.PyTree as G

a = G.cartHexa((0.,0.,0.), (0.1,0.1,0.1), (5,5,5))
```

```
a = C.convertArray2NGon(a); a = G.close(a)
#C.convertArrays2File([a], 'a.plt')
b = G.cartHexa((0.,0.,0.), (0.005,0.005,0.005), (5,5,5))
#C.convertArrays2File([b], 'b.plt')

m = XOR.adaptCells(a,b, sensor_type=0)
m = XOR.closeOctalCells(m)
C.convertPyTree2File(m, 'out.cgns')

m = XOR.adaptCells(a,b, sensor_type=1)
m = XOR.closeOctalCells(m)
C.convertPyTree2File(m, 'xout.cgns')
```

Tips and Notes:

- Do this transformation before calling any Volume-Volume boolean operations in order to improve the mesh quality of the result.
- When the input mesh has mixed-basic-type elements, only Tets and Hexas will be considered currently for adaptation. but the result wil be conformal, the pyramids and prisms will modified to respect the conformity.

Intersector.**adaptBox**(*a*, *box_ratio*)

Adapts the bounding box of a cloud of points. Adaptation is an octal 2:1 decomposition.

Parameters

- **a** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Input points cloud
- **box_ratio** – ratio to scale the box

Example of use:

- adaptBox (array):

```
# - adapt the bounding box of a point cloud (array) -

import Converter as C
import Generator as G
import Intersector as XOR

a = G.cartHexa((0.,0.,0.), (0.1,0.1,0.1), (5,5,5))
a = C.convertArray2NGon(a); a = G.close(a)

m = XOR.adaptBox(a, box_ratio=10.)
```

```
m = XOR.closeOctalCells(m) # optional : to close the polyhedral cells

C.convertArrays2File([m], 'out.plt')
```

- adaptBox (pyTree):

```
# - adapt the bounding box of a point cloud (array) -

import Converter.PyTree as C
import Generator.PyTree as G
import Intersector.PyTree as XOR

a = G.cartHexa((0.,0.,0.), (0.1,0.1,0.1), (5,5,5))
a = C.convertArray2NGon(a); a = G.close(a)

m = XOR.adaptBox(a, box_ratio=10.)

m = XOR.closeOctalCells(m) # optional : to close the polyhedral cells

C.convertPyTree2File(m, 'out.cgns')
```

3.3 Extraction Functions

Intersector.**extractPathologicalCells**(*a*, *neigh_level=0*)

Extracts cells that will potentially cause a failure when running a CFD solver. There are 4 zones upon exit, one for each pathology:

- Non-centroid-star-shaped Cells
- Cells having degenrated polygons for which the normal cannot be computed
- Cells having degenrated polygons for a delaunay triangulation fails
- Open Cells

Parameters

- **a** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Input mesh
- **neigh_level** (int) – Number of neighbor layers (surounding pathologies) to extract as well

Example of use:

- extractPathologicalCells (array):

```
# - Extract pathological cells (uncomputable or non-star) - (array)

import Converter as C
import Intersector as XOR

M1 = C.convertFile2Arrays('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1[0])

M2 = C.convertFile2Arrays('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2[0])

tol = -0.5e-3

m = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=1)
#C.convertArrays2File([m], 'i.plt')

m=XOR.extractPathologicalCells(m, 2) # ask for 2 level of neighgbors

C.convertArrays2File(m, 'out.plt')
```

- extractPathologicalCells (pyTree):

```
# - Extract pathological cells (PyTree) -
# uncomputable or non-star
import Converter.PyTree as C
import Intersector.PyTree as XOR

M1 = C.convertFile2PyTree('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1)

M2 = C.convertFile2PyTree('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2)

tol = -0.5e-3

t = XOR.booleanMinus(M1, M2, tol, preserve_right=0, solid_right=0, agg_mode=1)

t = XOR.extractPathologicalCells(t, 2) # ask for 2 level of neighgbors

C.convertPyTree2File(t, "out.cgns")
```

`Intersector.`**`extractOuterLayers`**(*a*, *N*, *discard_external=0*)

Extracts prescribed outer cell layers.

Parameters

- **a** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Input mesh
- **N** (int) – Number of layers to extract
- **discard_external** (0 or 1) – For volume mesh with holes (e.g. external flow), set it to 1 to extract only layers around bodies, or 0 to extract over all the outer polygons.

Example of use:

- extractOuterLayers (array):

```
# - extractOuterLayers (array) -
import Converter as C
import Intersector as XOR
import Generator as G

M1 = C.convertFile2Arrays('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1[0])
M1 = C.conformizeNGon(M1); M1 = G.close(M1)
m = XOR.extractOuterLayers(M1, 1, discard_external=0)

C.convertArrays2File(m, "out.plt")
```

- extractOuterLayers (pyTree):

```
# - extractOuterLayers (pyTree) -
import Converter.PyTree as C
import Intersector.PyTree as XOR
import Generator.PyTree as G
t = C.convertFile2PyTree('boolNG_M1.tp')
t = C.conformizeNGon(t); t = G.close(t)
t = XOR.extractOuterLayers(t, 1, discard_external=0)

C.convertPyTree2File(t, "out.cgns")
```

`Intersector.`**`getOverlappingFaces`**(*t1*, *t2*, *RTOL*, *ps_min*, *dir2*)

Detects all the overlapping polygons in t1 and t2. Returns the result as a list sized as the number of zones in t1. Each entry gives 2 lists : the first is the pg ids in t1 ith-zone, the second is the pg ids in t2 (joined).

Parameters

- **t1** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Input mesh
- **t2** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Input mesh
- **RTOL** (float) – relative tolerance
- **RTOL** – minimal dot product between normals of a pair of polygon to consider them as potentially overlapping.

Example of use:

- getOverlappingFaces (array):

```
import Generator as G
import Transform as T
import Converter as C
import Converter as I
import Intersector as XOR
import KCore.test as test
import Post as P

t1 = G.cart((0,0,0), (1,1,1), (10,10,10))
t1 = C.convertArray2NGon(t1); t1 = G.close(t1)
t2 = G.cart((1.,0,0), (1,1,1), (10,10,10))
t2 = C.convertArray2NGon(t2); t2 = G.close(t2)

# test 1 : volume/volume
res = XOR.getOverlappingFaces(t1, t2, RTOL = 0.05, ps_min = 0.95)

# create a list of polygon list (t1), one list per zone

t = XOR.agglomerateCellsWithSpecifiedFaces(t1, res[0])

C.convertArrays2File([t], "out.plt")

#test 2 : volume/surface

t2 = P.exteriorFaces(t2)
t2 = XOR.convertNGON2DToNGON3D(t2)

res = XOR.getOverlappingFaces(t1, t2, RTOL = 0.05, ps_min = 0.95)

t = XOR.agglomerateCellsWithSpecifiedFaces(t1, res[0])

C.convertArrays2File([t], "out1.plt")
```

- getOverlappingFaces (pyTree):

```python
import Generator.PyTree as G
import Transform.PyTree as T
import Converter.PyTree as C
import Converter.Internal as I
import Intersector.PyTree as XOR
import KCore.test as test
import Post.PyTree as P

t1 = G.cart((0,0,0), (1,1,1), (10,10,10))
t1 = C.convertArray2NGon(t1); t1 = G.close(t1)
t2 = G.cart((1.,0,0), (1,1,1), (10,10,10))
t2 = C.convertArray2NGon(t2); t2 = G.close(t2)

# test 1 : volume/volume
res = XOR.getOverlappingFaces(t1, t2, RTOL = 0.05, ps_min = 0.95)

# create a list of polygon list (t1), one list per zone
nb_zones = len(res)
t1zones_pgids = []
for i in range(nb_zones):
  t1zones_pgids.append(res[i][0])

t = XOR.agglomerateCellsWithSpecifiedFaces(t1, t1zones_pgids)

C.convertPyTree2File(t, "out.cgns")

#test 2 : volume/surface

t2 = P.exteriorFaces(t2)
t2 = XOR.convertNGON2DToNGON3D(t2)

res = XOR.getOverlappingFaces(t1, t2, RTOL = 0.05, ps_min = 0.95)

t1zones_pgids = []
for i in range(nb_zones):
  t1zones_pgids.append(res[i][0])

t = XOR.agglomerateCellsWithSpecifiedFaces(t1, t1zones_pgids)

C.convertPyTree2File(t, "out1.cgns")
```

Tips and Notes:

- When asembling 2 meshes m1 and m2 where m2 is priorized , to improve the assembly quality, do before calling the boolean union:

1. getOverlappingFaces (m1, skin(m2)) where skin(m2) is the external polygonal

skin of m2

2. agglomerateCellsWithSpecifiedFaces on m1 with the above list of polygons

---

3.4 Check Functions

`Intersector.`**`selfX`**`(a)`

Checks self-intersections in a mesh. Returns the first two cell indices that collide.

Parameters **a** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Input mesh

Example of use:

- selfX (array):

```
# - boolean difference (array) -
import Intersector as XOR
import Converter as C
import Transform as T

M1 = C.convertFile2Arrays('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1[0])
M1 = C.conformizeNGon(M1)
M1 = XOR.closeOctalCells(M1)


M2 = C.convertFile2Arrays('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2[0])
M2 = C.conformizeNGon(M2)
M2 = XOR.closeOctalCells(M2)

tol = -0.5e-3


M = T.join(M1,M2)
M = XOR.selfX(M)

C.convertArrays2File([M], 'out.plt')
```

- selfX (pyTree):

```
# - boolean difference (array) -
import Intersector.PyTree as XOR
```

```
import Converter.PyTree as C
import Transform.PyTree as T

M1 = C.convertFile2PyTree('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1)
M1 = C.conformizeNGon(M1)
M1 = XOR.closeOctalCells(M1)


M2 = C.convertFile2PyTree('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2)
M2 = C.conformizeNGon(M2)
M2 = XOR.closeOctalCells(M2)

tol = -0.5e-3


M = T.join(M1,M2)
M = XOR.selfX(M)

C.convertPyTree2File(M, 'out.cgns')
```

`Intersector.`**`diffMesh`**(*a1*, *a2*)

Extracts the diff between 2 meshes. Returns 2 zones : one zone with the a1 cells that are not in a2, the second one is the reciprocal.

Parameters

- **a1** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Input mesh
- **a2** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – Input mesh

Example of use:

- diffMesh (array):

```
# - boolean diffSurf (array) -
import Generator as G
import Geom as D
import Converter as C
import Intersector as XOR

# octree
s = D.sphere((0,0,0), 1., 100); snear = 0.1
t = G.octree([s],[snear], dfar=5., balancing=1,ratio=2)
```

```
# ngon conversion
t = C.convertArray2NGon(t)
# ngon conformization
t = C.conformizeNGon(t); t = G.close(t)
# ngon close cells
t = XOR.closeOctalCells(t)
#t = XOR.reorientExternalFaces(t)

# ngon conversion of the sphere
s = C.convertArray2NGon(s)
# ngon conversion to the nuga format
s = XOR.convertNGON2DToNGON3D(s)
#s = XOR.reorientExternalFaces(s)

# Boolean operation
x = XOR.diffSurf(t, s, tol = 0., preserve_right=1, agg_mode=2) # agg_mode=2 :␣
↪full mode aggregation

xa = XOR.agglomerateSmallCells(x, 0., 10.)

x = XOR.diffMesh(x,xa[0])
C.convertArrays2File(x, 'diffM.plt')
```

- diffMesh (pyTree):

```
# - boolean diffSurf (array) -
import Generator.PyTree as G
import Geom.PyTree as D
import Converter.PyTree as C
import Intersector.PyTree as XOR

# octree
s = D.sphere((0,0,0), 1., 100); snear = 0.1
t = G.octree([s],[snear], dfar=5., balancing=1,ratio=2)

# ngon conversion
t = C.convertArray2NGon(t)
# ngon conformization
t = C.conformizeNGon(t); t = G.close(t)
# ngon close cells
t = XOR.closeOctalCells(t)
#t = XOR.reorientExternalFaces(t)

# ngon conversion of the sphere
```

```
s = C.convertArray2NGon(s)
# ngon conversion to the nuga format
s = XOR.convertNGON2DToNGON3D(s)
#s = XOR.reorientExternalFaces(s)

# Boolean operation
x = XOR.diffSurf(t, s, tol = 0., preserve_right=1, agg_mode=2) # agg_mode=2 :␣
↪full mode aggregation

xa = XOR.agglomerateSmallCells(x, 0., 10.)

x = XOR.diffMesh(x,xa)
C.convertPyTree2File(x, 'diffM.cgns')
```

Intersector.**checkCellsClosure**(*a*)

Checks that input mesh cells are closed, i.e. each cell' edge is shared by exactly two polygons.

Parameters **a** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Input mesh

Example of use:

- checkCellsClosure (array):

```
# - boolean reorientExternalFaces (array) -
import Generator as G
import Converter as C
import Intersector as XOR


a = G.cartHexa((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
a = C.convertArray2NGon(a)

err = XOR.checkCellsClosure(a)
```

- checkCellsClosure (pyTree):

```
# - boolean checkCellsClosure (array) -
import Generator.PyTree as G
import Converter.PyTree as C
import Intersector.PyTree as XOR
```

```
import Geom.PyTree as D

M1 = C.convertFile2PyTree('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1)

err = XOR.checkCellsClosure(M1)
```

Intersector.**computeAspectRatio**(*a*, *vmin=0.*)

For each cell, the aspect ratio with each of its neighbors is computed as the ratio of the biggest volume to the smallest one.

The maximum over all the neighbors is chosen:

Aspect Ratio for Cell i = MAX_k (MAX(vi, vk) / MIN(vi, vk)) where k is a neighbor.

Parameters

- **a** ([array, list of arrays] or [pyTree, base, zone, list of zones]) – Input mesh
- **vmin** (float) – volume threshold.

Example of use:

- computeAspectRatio (array):

```
# - Extract pathological cells (uncomputable or non-star) (array) -
import Converter as C
import Intersector as XOR

M1 = C.convertFile2Arrays('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1[0])

M2 = C.convertFile2Arrays('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2[0])

tol = -0.5e-3

m = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=1)

aspect_ratio = XOR.computeAspectRatio(m)
aspect_ratio = C.center2Node(aspect_ratio)

C._addVars([m, aspect_ratio])

C.convertArrays2File(m, 'out.plt')
```

- computeAspectRatio (pyTree):

```
# - Extract pathological cells (uncomputable or non-star) - (array)

import Converter.PyTree as C
import Intersector.PyTree as XOR

M1 = C.convertFile2PyTree('boolNG_M1.tp')
M1 = C.convertArray2NGon(M1)

M2 = C.convertFile2PyTree('boolNG_M2.tp')
M2 = C.convertArray2NGon(M2)

tol = -0.5e-3

t = XOR.booleanMinus(M1, M2, tol, preserve_right=1, solid_right=1, agg_mode=1)

t=XOR.computeAspectRatio(t)
C.convertPyTree2File(t, 'out.cgns')
```

3.5 Conversion Functions

Intersector.**convertNGON2DToNGON3D**(*a*)

Converts a polygon surface stored in the Cassiopee NGON format (Face/Edge) to a Face/Node format.

CHAPTER

FOUR

INDEX

- genindex
- modindex
- search