Generator Documentation

Release 2.9

/ELSA/MU-09022/V2.9

Jun 07, 2019

CONTENTS

CHAPTER

ONE

PREAMBLE

Generator module works on arrays (as defined in Converter) or on CGNS/python trees (pyTrees) containing grid information (coordinates must be defined).

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 Generator module:

```
import Generator as G
```

For use with the pyTree interface:

```
import Generator.PyTree as G
```

CHAPTER

TWO

LIST OF FUNCTIONS

– Basic grid generation

Generator.cart(Xo, H, N[, api])	Create a cartesian mesh defined by a structured array.
Generator.cartHexa(Xo, H, N[, api])	Create a cartesian mesh defined by an hexaedrical array.
Generator.cartTetra(Xo, H, N[, api])	Create a cartesian mesh defined by a tetraedrical array.
Generator.cartPenta(Xo, H, N[, api])	Create a cartesian mesh defined by a prismatic array.
Generator.cartPyra(Xo, H, N[, api])	Create a cartesian mesh defined by a pyramidal array.
Generator.cartNGon(Xo, H, N[, api])	Create a cartesian mesh defined by a NGON array.
Generator.cylinder(Xo, R1, R2, tetas, tetae, ...)	Create a portion of regular cylindrical grid.
Generator.cylinder2(Xo, R1, R2, tetas, ...)	Create a portion of cylindrical grid.
Generator.cylinder3(arrayxz, tetas, tetae, ...)	Create a portion of cylindrical grid.

– General purpose grid generators

Generator.delaunay(array[, tol, keepBB])	Create a delaunay mesh given a set of points defined by array.
Generator.constrainedDelaunay(cont0[, tol, ...])	Create a constrained-Delaunay mesh starting from a BAR-array defining the contour.
Generator.checkDelaunay(contour, tri)	Check if the Delaunay triangulation defined by tri is inside the contour.
Generator.T3mesher2D(a[, grading, ...])	Create a delaunay mesh given a set of points defined by a.

Continued on next page

Table 2.2 – continued from previous page

Generator.tetraMesher(a[, maxh, grading, ...])	Create a TRI/TETRA mesh given a set of BAR or surfaces in a.
Generator.TFI(arrays)	Generate a transfinite interpolation mesh from boundaries.
Generator.TFITri(a1, a2, a3)	Generate a transfinite interpolation mesh from 3 input curves.
Generator.TFIO(a)	Generate a transfinite interpolation mesh for 1 input curve.
Generator.TFIHalfO(a1, a2)	Generate a transfinite interpolation mesh for 2 input curves.
Generator.TFIMono(a1, a2)	Generate a transfinite interpolation mesh for 2 input curves.
Generator.hyper2D(array, arrayd, type)	Generate an hyperbolic mesh.
Generator.PolyLine.polyLineMesher(polyLine, ...)	Generate a multiple mesh for a polyline.
Generator.PolyC1.polyC1Mesher(curve, h, ...)	Generate a multiple mesh for a C1 curve (polyC1).
Generator.pointedHat(array, coord)	Create a structured surface defined by a contour and a point (x,y,z).
Generator.stitchedHat(array, offset[, tol, tol2])	Create a structured surface defined by a contour and an offset (dx,dy,dz).
Generator.surfaceWalk(surfaces, c, distrib)	Generate a surface mesh by a walk on a list of surfaces, starting from a contour c and following constraints.
Generator.collarMesh(s1, s2, distribj, distribk)	Generates a collar mesh starting from s1 and s2 surfaces, distributions along the surfaces and along the normal direction, with respect to the assembly type between grids.

– Cartesian grid generators

Generator.gencartmb(bodies, h, Dfar, nlvl)	
Generator.octree(stlArrays[, snearList, ...])	Generate an octree (or a quadtree) mesh starting from a list of TRI (or BAR) arrays defining bodies, a list of corresponding snears, and the extension dfar of the mesh.
Generator.octree2Struct(a[, vmin, ext, ...])	Generates a structured set of regular Cartesian grid starting from an octree HEXA or QUAD mesh.
Generator.adaptOctree(octreeHexa, indicField)	Adapt an unstructured octree w.r.t.

Continued on next page

Table 2.3 – continued from previous page

Generator.expandLayer(octreeHexa[, level, ...])	Expand the layer of octree elements of level l of one additional layer.

– Operations on meshes

Generator.close(array[, tol])	Close a mesh defined by an array gathering points closer than tol.
Generator.selectInsideElts(array, curvesList)	Select elements whose center is in the surface delimited by curves.
Generator.map(array, d[, dir])	Map a distribution on a curve or a surface.
Generator.mapSplit(array, dist[, splitCrit, ...])	Split a curve and map a distribution on the set of split curves.
Generator.refine(array, power[, dir])	Refine a mesh of power power along all the directions or on a specified one.
Generator.mapCurvature(array, N, power, dir)	
Generator.densify(array, h)	
Generator.grow(array, vector)	Grow a surface array of one layer by deplacing points of vector.
Generator.stack(array1[, array2])	Stack two meshes (with same nixnj) into a single mesh.
Generator.addNormalLayers(surface, distrib)	Generate N layers to a surface following normals.
Generator.TTM(array[, niter])	Smooth a mesh with Thompson-Mastin elliptic generator.
Generator.snapFront(meshes, surfaces[, ...])	Adapt meshes to a given surface (cellN defined).
Generator.snapSharpEdges(meshes, surfaces[, ...])	Adapt meshes to a given surface sharp edges.

– Operations on surface meshes

Generator.fittingPlaster(contour[, bumpFactor])	Generate a structured points cloud over a BAR.
Generator.gapfixer(contour, cloud[, ...])	Fix a gap defined by a contour bar and a point cloud representing the gap surface.
Generator.gapsmanager(components[, mode, ...])	Fix a gap between several component surfaces (list of arrays).
Generator.quad2Pyra(array[, hratio])	Create a set of pyramids from a set of quads.

– Information on generated meshes

Generator.barycenter(array[, weight])	Get the barycenter of an array.
Generator.bbox(arrays)	Returns the bounding box of a list of arrays.
Generator.bboxOfCells(array)	Return the bounding box of all cells of an array.
Generator.BB(array[, method, weighting])	Return the axis-aligned or oriented bounding box of an array as an array.
Generator.CEBBIntersection(array1, array2[, tol])	Get the Cartesian Elements bounding box intersection of 2 arrays.
Generator.bboxIntersection(array1, array2[, ...])	Return the intersection of bounding boxes of 2 arrays.
Generator.checkPointInCEBB(array, P)	Check if point P is in the Cartesian Elements Bounding Box of array.
Generator.getVolumeMap(array)	Return the volume map in an array.
Generator.getNormalMap(array)	Return the map of surface normals in an array.
Generator.getSmoothNormalMap(array[, niter, eps])	Return the map of smoothed and non-normalized surface normals in an array.
Generator.getOrthogonalityMap(array)	Return the orthogonality map in an array.
Generator.getRegularityMap(array)	Return the regularity map in an array.
Generator.getTriQualityMap(array)	Return a TRI quality measure map in an array.
Generator.getCellPlanarity(array)	Return the cell planarity of a surface mesh in an array.
Generator.getCircumCircleMap(array)	Return the map of circum circle radius of any cell in a TRI array.
Generator.getInCircleMap(array)	Return the map of inscribed circle radius of any cell in a TRI array.
Generator.getEdgeRatio(array[, dim])	Computes the ratio between the max and min lengths of all the edges of cells in an array.
Generator.getMaxLength(array[, dim])	Computes the max length of all the edges of cells in an array.

– Operations on distributions

Generator.enforceX(array, x0, enforcedh, N)	Enforce a x0-centered line in a distribution defined by an array.
Generator.enforceMoinsX(array, enforcedh, N)	Enforce the last X-line in a distribution (one sided, left).
Generator.enforcePlusX(array, enforcedh, N)	Enforce the first X-line in a distribution defined by an array.
Generator.enforceLine(array, arrayline, ...)	Enforce a line in a distribution.

Continued on next page

Table 2.7 – continued from previous page

Generator.enforcePoint(array, x0)	Enforce a point in a distribution.
Generator.enforceCurvature(arrayD, arrayC[, ...])	Enforce curvature of a curve in a distribution.
Generator.addPointInDistribution(array, ind)	Add a point in a distribution defined by array.

CHAPTER

THREE

CONTENTS

3.1 Basic grid generation

Generator.**cart**(*(xo, yo, zo)*, *(hi, hj, hk)*, *(ni, nj, nk)*)

Create a structured Cartesian mesh with ni x nj x nk points starting from point (xo,yo,zo) and of step (hi,hj,hk).

Parameters

- **(xo,yo,zo)** (3-tuple of floats) – coordinates of the starting point
- **(hi,hj,hk)** (3-tuple of floats) – values of advancing step in the three directions
- **(ni,nj,nk)** (3-tuple of integers) – number of points in each direction

Returns a 1D, 2D or 3D structured mesh

Return type array or pyTree zone

Example of use:

- Cartesian mesh generation (array):

```
# - cart (array) -
import Converter as C
import Generator as G
a = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
C.convertArrays2File(a, 'out.plt')
```

- Cartesian mesh generation (pyTree):

```
# - cart (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
```

```
a = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,11,12))
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**cartHexa**(*(xo, yo, zo)*, *(hi, hj, hk)*, *(ni, nj, nk)*)

Create an unstructured hexahedral mesh defined from a Cartesian grid of ni x nj x nk points starting from point (xo,yo,zo) and of step (hi,hj,hk). Type of elements are 'QUAD' for 2D arrays and 'HEXA' for 3D arrays.

Parameters

- **(xo,yo,zo)** (3-tuple of floats) – coordinates of the starting point
- **(hi,hj,hk)** (3-tuple of floats) – values of advancing step in the three directions
- **(ni,nj,nk)** (3-tuple of integers) – number of points in each direction

Returns a 1D, 2D or 3D unstructured mesh

Return type array or pyTree zone

Example of use:

- Cartesian hexa mesh generation (array):

```
# - cartHexa (array) -
import Generator as G
import Converter as C

a = G.cartHexa((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
C.convertArrays2File([a], 'out.plt')
```

- Cartesian hexa mesh generation (pyTree):

```
# - cartHexa (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C

a = G.cartHexa((0.,0.,0.), (0.1,0.1,0.2), (10,10,1))
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**cartTetra**(*(xo, yo, zo)*, *(hi, hj, hk)*, *(ni, nj, nk)*)

Create an unstructured tetrahedral mesh defined from a Cartesian grid of ni x nj x

nk points starting from point (xo,yo,zo) and of step (hi,hj,hk). Type of elements are 'TRI' for 2D arrays and 'TETRA' for 3D arrays.

Parameters

- **(xo,yo,zo)** (3-tuple of floats) – coordinates of the starting point
- **(hi,hj,hk)** (3-tuple of floats) – values of advancing step in the three directions
- **(ni,nj,nk)** (3-tuple of integers) – number of points in each direction

Returns a 1D, 2D or 3D unstructured mesh

Return type array or pyTree zone

Example of use:

- Cartesian tetra mesh generation (array):

```
# - cartTetra (array) -
import Generator as G
import Converter as C
a = G.cartTetra((0.,0.,0.), (0.1,0.1,0.2), (10,10,1))
C.convertArrays2File(a, 'out.plt')
```

- Cartesian tetra mesh generation (pyTree):

```
# - cartTetra (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C

a = G.cartTetra((0.,0.,0.), (0.1,0.1,0.2), (10,10,1))
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**cartPenta**(*(xo, yo, zo), (hi, hj, hk), (ni, nj, nk)*)

Create an unstructured prismatic mesh defined from a regular Cartesian mesh. The initial Cartesian mesh is defined by ni x nj x nk points starting from point (xo,yo,zo) and of step (hi,hj,hk). Type of elements is 'PENTA'.

Parameters

- **(xo,yo,zo)** (3-tuple of floats) – coordinates of the starting point
- **(hi,hj,hk)** (3-tuple of floats) – values of advancing step in the three directions
- **(ni,nj,nk)** (3-tuple of integers) – number of points in each direction

Returns a 1D, 2D or 3D unstructured mesh

Return type array or pyTree zone

Example of use:

- Cartesian penta mesh generation (array):

```
# - cartPenta (array) -
import Generator as G
import Converter as C
a = G.cartPenta((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
C.convertArrays2File([a], "out.plt")
```

- Cartesian penta mesh generation (pyTree):

```
# - cartPenta (pyTree)-
import Generator.PyTree as G
import Converter.PyTree as C

a = G.cartPenta((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**cartPyra**(*(xo, yo, zo)*, *(hi, hj, hk)*, *(ni, nj, nk)*)

Create an unstructured pyramidal mesh defined from a regular Cartesian mesh. The initial Cartesian mesh is defined by ni x nj x nk points starting from point (xo,yo,zo) and of step (hi,hj,hk). Type of elements is 'PYRA'.

Parameters

- **(xo,yo,zo)** (3-tuple of floats) – coordinates of the starting point
- **(hi,hj,hk)** (3-tuple of floats) – values of advancing step in the three directions
- **(ni,nj,nk)** (3-tuple of integers) – number of points in each direction

Returns a 1D, 2D or 3D unstructured mesh

Return type array or pyTree zone

Example of use:

- Cartesian pyra mesh generation (array):

```
# - cartHexa (array) -
import Generator as G
import Converter as C
```

```
a = G.cartPyra((0.,0.,0.), (1,1,1), (20,20,20))
C.convertArrays2File(a, 'out.tp')
```

- Cartesian pyra mesh generation (pyTree):

```
# - cartPyra (pyTree)-
import Generator.PyTree as G
import Converter.PyTree as C

a = G.cartPyra((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**cartNGon**(*(xo, yo, zo)*, *(hi, hj, hk)*, *(ni, nj, nk)*)

Create a NGON mesh defined from a regular Cartesian mesh. The initial Cartesian mesh is defined by ni x nj x nk points starting from point (xo,yo,zo) and of step (hi,hj,hk). Type of elements is 'NGON'.

Parameters

- **(xo,yo,zo)** (3-tuple of floats) – coordinates of the starting point
- **(hi,hj,hk)** (3-tuple of floats) – values of advancing step in the three directions
- **(ni,nj,nk)** (3-tuple of integers) – number of points in each direction

Returns a 1D, 2D or 3D unstructured mesh

Return type array or pyTree zone

Example of use:

- Cartesian ngon mesh generation (array):

```
# - cartNGon (array) -
import Generator as G
import Converter as C

a = G.cartNGon((0.,0.,0.), (0.1,0.1,0.2), (20,20,20))
C.convertArrays2File(a, 'out.plt')
```

- Cartesian ngon mesh generation (pyTree):

```
# - cartNGon (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C

a = G.cartNGon((0.,0.,0.), (0.1,0.1,0.2), (2,2,2))
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**cylinder**(*(xo, yo, zo)*, *R1*, *R2*, *tetas*, *tetae*, *H*, *(ni, nj, nk)*)

Create a regular cylindrical grid (or a portion of cylinder between tetas and tetae) with ni x nj x nk points, of center-bottom point (xo,yo,zo), of inner radius R1, outer radius R2 and height H. For a direct mesh, use tetae < tetas.

Parameters

- **(xo,yo,zo)** (3-tuple of floats) – coordinates of the starting point
- **R1** (float) – value of inner radius
- **R2** (float) – value of outer radius
- **tetas** (float) – start angle (in degree)
- **tetae** (float) – end angle (in degree)
- **(ni,nj,nk)** (3-tuple of integers) – number of points in each direction

Returns a 3D structured mesh

Return type array or pyTree zone

Example of use:

- Regular cylinder mesh generation (array):

```
# - cylinder (array) -
import Generator as G
import Converter as C
a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 10., (50,50,30))
C.convertArrays2File([a], "out.plt")
```

- Regular cylinder mesh generation (pyTree):

```
# - cylinder (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 10., (50,50,30))
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**cylinder2**(*(xo, yo, zo)*, *R1*, *R2*, *tetas*, *tetae*, *H*, *arrayR*, *arrayTeta*, *arrayZ*)

Create an irregular cylindrical grid (or a portion of cylinder between tetas and tetae) with ni x nj x nk points, of center-bottom point (xo,yo,zo), of inner radius R1, outer radius R2, height H and with distributions in r, teta, z. Distributions are arrays defining 1D meshes (x and i varying) giving a distribution in [0,1]. Their number of points gives ni, nj, nk.

Parameters

- **(xo,yo,zo)** (3-tuple of floats) – coordinates of the starting point
- **R1** (float) – value of inner radius
- **R2** (float) – value of outer radius
- **tetas** (float) – start angle (in degree)
- **tetae** (float) – end angle (in degree)
- **H** (float) – value of cylinder height
- **arrayR** (array) – distribution along radius
- **arrayTeta** (array) – distribution along azimuth
- **arrayZ** (array) – distribution along height

Returns a 3D structured mesh

Return type array or pyTree zone

Example of use:

- Irregular cylinder mesh generation (array):

```
# - cylinder2 (array) -
import Converter as C
import Generator as G
r = G.cart((0.,0.,0.), (0.1, 1., 1.), (11, 1, 1))
teta = G.cart((0.,0.,0.), (0.1, 1., 1.), (11, 1, 1))
z = G.cart((0.,0.,0.), (0.1, 1., 1.), (11, 1, 1))
cyl = G.cylinder2( (0.,0.,0.), 0.5, 1., 360., 0., 10., r, teta, z)
C.convertArrays2File([cyl], "out.plt")
```

- Irregular cylinder mesh generation (pyTree):

```
# - cylinder2 (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
```

```
r = G.cart((0.,0.,0.), (0.1, 1., 1.), (11, 1, 1))
teta = G.cart((0.,0.,0.), (0.1, 1., 1.), (11, 1, 1))
z = G.cart((0.,0.,0.), (0.1, 1., 1.), (11, 1, 1))

cyl = G.cylinder2( (0.,0.,0.), 0.5, 1., 360., 0., 10., r, teta, z)
C.convertPyTree2File(cyl, 'out.cgns')
```

Generator.**cylinder3**(*a*, *tetas*, *tetae*, *arrayTeta*)

Create an irregular cylindrical grid (or a portion of cylinder between tetas and tetae) from a xz plane mesh defined by a and a teta distribution defined by arrayTeta.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones, base, pyTree]) – definition of the xz plane mesh
- **tetas** (float) – start angle (in degree)
- **tetae** (float) – end angle (in degree)
- **arrayTeta** (array) – distribution along azimuth

Returns a 3D structured mesh

Return type array or pyTree zone

Example of use:

- Irregular cylinder mesh generation from a xz plane (array):

```
# - cylinder3 (array) -
import Generator as G
import Converter as C
teta = G.cart((0.,0.,0.), (0.1, 1., 1.), (11, 1, 1))
xz = G.cart((0.1,0.,0.), (0.1,1.,0.2), (20, 1, 30))
cyl = G.cylinder3( xz, 0., 90., teta)
C.convertArrays2File([cyl], 'out.plt')
```

- Irregular cylinder mesh generation from a xz plane (pyTree):

```
# - cylinder3 (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
teta = G.cart((0.,0.,0.), (0.1, 1., 1.), (11, 1, 1))
xz = G.cart((0.1,0.,0.), (0.1,1.,0.2), (20, 1, 30))
cyl = G.cylinder3(xz, 0., 90., teta)
C.convertPyTree2File(cyl, 'out.cgns')
```

3.2 General purpose grid generator

Generator.**delaunay**(*a*, *tol=1.e-10*, *keepBB=0*)

Create a 2D Delaunay type mesh from an array. The array can be a 2D structured array, or an unstructured array of type 'NODE', 'TRI' or 'QUAD'. Tol is a geometric tolerance. Points nearer than tol are merged. If keepBB is set to 1, the bounding box is kept in the final triangulation.

Parameters

- **a** (`[array] or [zone]`) – structured or unstructured 2D mesh
- **tol** (`float`) – geometric tolerance
- **keepBB** (`integer (0 or 1)`) – keep bounding box in result?

Returns a 2D unstructured mesh

Return type Identical to a

Example of use:

- 2D Delaunay mesh generation (array):

```
# - delaunay (array) -
import Generator as G
import Converter as C
ni = 11; nj = 11; nk = 1
hi = 1./(ni-1); hj = 1./(nj-1); hk = 1.
a = G.cart((0.,0.,0.), (hi,hj,hk), (ni,nj,nk))
b = G.delaunay(a)
C.convertArrays2File([a,b], "out.plt")
```

- 2D Delaunay mesh generation (pyTree):

```
# - delaunay (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G

ni = 11; nj = 11; nk = 1
hi = 1./(ni-1); hj = 1./(nj-1); hk = 1.
a = G.cart((0.,0.,0.), (hi,hj,hk), (ni,nj,nk))
b = G.delaunay(a); b[0] = 'delaunay'
t = C.newPyTree(['Base', 2, a,b])
C.convertPyTree2File(t, 'out.cgns')
```

Generator.**constrainedDelaunay**(*c*, *tol=1.e-10*, *keepBB=0*)

Create a constrained Delaunay triangulation of the convex hull of a contour c. Contour must be a BAR-array and must be in the plane (x,y). Tol is a geometric tolerance. Points nearer than tol are merged. If keepBB is set to 1, the bounding box is kept in the final triangulation.

Parameters

- **c** (BAR-array) – contour in xy plane
- **tol** (float) – geometric tolerance
- **keepBB** (integer (0 or 1)) – keep bounding box in result?

Returns a 2D unstructured mesh

Return type Identical to a

Example of use:

- 2D Delaunay mesh generation from a contour (array):

```
# - constrainedDelaunay (array) -
import Converter as C
import Generator as G
import Transform as T
import Geom as D

A = D.text1D('CASSIOPEE')
A = C.convertArray2Tetra(A); a = T.join(A)
# Triangulation respecting given contour
tri = G.constrainedDelaunay(a)
C.convertArrays2File([a,tri], "out.plt")
```

- 2D Delaunay mesh generation from a contour (pyTree):

```
# - constrainedDelaunay (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Geom.PyTree as D
import Transform.PyTree as T

A = D.text1D('CASSIOPEE')
A = C.convertArray2Tetra(A); a = T.join(A)
# Triangulation respecting given contour
tri = G.constrainedDelaunay(a)
C.convertPyTree2File(tri, 'out.cgns')
```

Generator.**checkDelaunay**(*c*, *tri*)

Check if the Delaunay triangulation defined in tri is inside the contour c.

Parameters

- **c** (BAR-array) – contour in xy plane
- **tri** (array or pyTree) – 2D Delaunay triangulation mesh

Returns contour

Return type BAR-array

Example of use:

- Check Delaunay triangulation wrt. contour (array):

```
# - checkDelaunay (array) -
import Converter as C
import Generator as G
import Transform as T
import Geom as D

A = D.text1D('CASSIOPEE')
A = C.convertArray2Tetra(A); a = T.join(A)
# Triangulation respecting given contour
tri = G.constrainedDelaunay(a)
res = G.checkDelaunay(a, tri)
C.convertArrays2File([res], "out.plt")
```

- Check Delaunay triangulation wrt. contour (pyTree):

```
# - checkDelaunay (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Geom.PyTree as D
import Transform.PyTree as T

A = D.text1D('CASSIOPEE')
A = C.convertArray2Tetra(A); a = T.join(A)
# Triangulation respecting given contour
tri = G.constrainedDelaunay(a)
res = G.checkDelaunay(a, tri)
C.convertPyTree2File(res, "out.cgns")
```

Generator.**T3mesher2D**(*a*, *triangulateOnly=0*, *grading=1.2*, *metricInterpType=0*)

Creates a 2D Delaunay mesh given a BAR defined in a. If triangulateOnly=1 then

only points of a are triangulated, if triangulateOnly=0, then interior points are inserted.

The grading parameter allows to control the growth ratio of the mesh metric : a value greater(lesser) than 1. tends to produce a coarser (finer) mesh in the region far from the boundaries. A value equal to 1. provides a uniform mesh over the domain. This grading is related to the metric field, it is not the size ratio between two adjacent edges or triangles.

The metricInterpType parameter controls the metrics interpolation type: either linear or geometric. A geometric metric interpolation tends to promote smaller sizes.

Parameters

- **c** (`[array, list of arrays] or [pyTree, base, zone, list of zones]`) – BAR-contour (soup of conformal edges defining an enclosed 2D-domain, can be non-manifold, i.e. having inner edges or subdomains)
- **triangulateOnly** (`integer (0 or 1)`) – insertion or not of interior points for the triangulation
- **grading** (`float (strictly positive value)`) – metric growth ratio
- **metricInterpType** (`integer (0 or 1)`) – metric interpolation type, linear(0) or geometric(1)

Returns 2D mesh

Return type Identical to input

Example of use:

- 2D Delaunay mesh generation from a BAR (array):

```
# - T3mesher2D (array) -
import Generator as G
import Converter as C
import Geom as D

a = D.circle((0,0,0), 1, N=50)
a = C.convertArray2Tetra(a)
a = G.close(a)
b = G.T3mesher2D(a, triangulateOnly=0, grading=1.2, metricInterpType=0) #␣
↪linear metric interpolation
C.convertArrays2File([a,b], 'outL.plt')

b = G.T3mesher2D(a, triangulateOnly=0, grading=1.2, metricInterpType=1) #␣
↪geometric metric interpolation
C.convertArrays2File([a,b], 'outG.plt')
```

- 2D Delaunay mesh generation from a BAR (pyTree):

```
# - T3mesher2D (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C
import Geom.PyTree as D

a = D.circle((0,0,0), 1, N=50)
a = C.convertArray2Tetra(a)
a = G.close(a)

b = G.T3mesher2D(a, triangulateOnly=0, grading=1.2, metricInterpType=0) #␣
↪linear metric interpolation
C.convertPyTree2File(b, 'outL.cgns')

b = G.T3mesher2D(a, triangulateOnly=0, grading=1.2, metricInterpType=1) #␣
↪geometric metric interpolation
C.convertPyTree2File(b, 'outG.cgns')
```

Generator.**tetraMesher**(*a*, *algo=1*)

Create a 3D tetra mesh given a TRI surface defined in a. If the TRI surface has external normals, tetras are filled inside the surface. If algo=0, netgen is used, if algo=1, tetgen is used.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones, base, pyTree]) – triangulated surface mesh
- **algo** (integer (0 or 1)) – choice parameter between netgen and tetgen

Returns 3D mesh

Return type [array] or [pyTree zone]

Example of use:

- 3D tetra mesh generation (array):

```
# - tetraMesher (array) -
import Generator as G
import Converter as C
import Post as P
import Transform as T

a = G.cart((0,0,0), (1,1,1), (3,3,3))
ext = P.exteriorFaces(a)
```

```
ext = C.convertArray2Tetra(ext)
ext = G.close(ext)
ext = T.reorder(ext, (-1,))
m = G.tetraMesher(ext, algo=1)
C.convertArrays2File(m, 'out.plt')
```

- 3D tetra mesh generation (pyTree):

```
# - tetraMesher (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C
import Post.PyTree as P
import Transform.PyTree as T

a = G.cart((0,0,0), (1,1,1), (3,3,3))
ext = P.exteriorFaces(a)
ext = C.convertArray2Tetra(ext)
ext = G.close(ext)
ext = T.reorder(ext, (-1,))
m = G.tetraMesher(ext, algo=1)
C.convertPyTree2File(m, 'out.cgns')
```

Generator.**TFI**([*imin*, *imax*, *jmin*, *jmax*, *kmin*, *kmax*])

Generate a mesh by transfinite interpolation (TFI). Generated mesh can be 2D or 3D structured, or unstructured TRI or PENTA mesh. Warning: the boundaries can be in a different order from the examples below, except for the PENTA TFI meshes. 2D structured mesh is built from imin, imax, jmin, jmax boundaries. 3D structured mesh is built from imin, imax, jmin, jmax, kmin, kmax boundaries. Dimensions must be equal for each pair (imin,imax), (jmin,jmax)... TRI mesh is built from imin, jmin, diag boundaries. Each boundary is a structured array with the same dimension. PENTA mesh is built from Tmin, Tmax triangles boundary and imin, imax, diag boundaries. Tmin, Tmax must be structured triangles of dimension nxn. imin, jmin, diag must be structured n*p arrays.

Parameters

- **imin** (array) – I-direction minimum boundary
- **imax** (array) – I-direction maximum boundary
- **jmin** (array) – J-direction minimum boundary
- **jmax** (array) – J-direction maximum boundary
- **kmin** (array) – K-direction minimum boundary
- **kmax** (array) – K-direction maximum boundary

- **diag** (array) – third direction boundary for TRI or PENTA meshes

Returns 2D or 3D mesh

Return type array or pyTree

Example of use:

- TFI mesh generation (array):

```
# - TFI 2D structured (array)
# - TFI 3D structured (array)
# - TFI TRI (array)
import Converter as C
import Generator as G
import Geom as D
import Transform as T

#------------------
# TFI 2D structured
#------------------
P0 = (0,0,0); P1 = (5,0,0); P2 = (0,7,0); P3 = (5,7,0)

# Geometrie
d1 = D.line(P0, P1); d2 = D.line(P2, P3)

pts = C.array('x,y,z',5,1,1)
x = pts[1][0]; y = pts[1][1]; z = pts[1][2]
x[0] = 0.; y[0] = 0.; z[0] = 0.
x[1] =-2.; y[1] = 2.; z[1] = 0.
x[2] =-3.; y[2] = 3.; z[2] = 0.
x[3] = 2.; y[3] = 5.; z[3] = 0.
x[4] = 0.; y[4] = 7.; z[4] = 0.
b1 = D.bezier(pts)

pts = C.array('x,y,z',5,1,1)
x = pts[1][0]; y = pts[1][1]; z = pts[1][2]
x[0] = 5.; y[0] = 0.; z[0] = 0.
x[1] = 3.; y[1] = 2.; z[1] = 0.
x[2] = 2.; y[2] = 3.; z[2] = 0.
x[3] = 6.; y[3] = 5.; z[3] = 0.
x[4] = 5.; y[4] = 7.; z[4] = 0.
b2 = D.bezier( pts )

C.convertArrays2File([d1, d2, b1, b2], "geom.plt")

# Regular discretisation of each line
Ni = 20
Nj = 10
```

```
r = G.cart((0,0,0), (1./(Ni-1),1,1), (Ni,1,1))
q = G.cart((0,0,0), (1./(Nj-1),1,1), (Nj,1,1))
r1 = G.map(d1, r)
r2 = G.map(d2, r)
r3 = G.map(b1, q)
r4 = G.map(b2, q)

# TFI 2D
m = G.TFI([r1, r2, r3, r4])
C.convertArrays2File([r1,r2,r3,r4,m], 'tfi2d.plt')

#------------------
# TFI 3D structured
#------------------
xo = 0.; yo = 0.; zo = 0.
nx = 21; ny = 21; nz = 21
hx = 1./(nx-1); hy = 1./(ny-1); hz = 1./(nz-1)

# z = cste
fzmin = G.cart((xo,yo,zo), (hx,hy,1.), (nx,ny,1))
fzmax = T.translate(fzmin, (0.,0.,1.))

# x = cste
fxmin = G.cart((xo,yo,zo),(1,hy,hz),(1,ny,nz))
fxmin = T.reorder(fxmin,(3,1,2))
fxmax = T.translate(fxmin, (1.,0.,0.))

# y = cste
fymin = G.cart((xo,yo,zo),(hx,1.,hz),(nx,1,nz))
fymin = T.reorder(fymin,(1,3,2))
fymax = T.translate(fymin, (0.,1.,0.))

r = [fxmin,fxmax,fymin,fymax,fzmin,fzmax]
m = G.TFI(r)
C.convertArrays2File(r+[m], 'tfi3d.plt')

#---------
# TFI TRI
#---------
l1 = D.line((0,0,0),(0,1,0), 15)
l2 = D.line((0,0,0),(1,0,0), 15)
l3 = D.line((1,0,0),(0,1,0), 15)
tri = G.TFI([l1,l2,l3])
C.convertArrays2File([tri], 'tfitri.plt')
```

- TFI mesh generation (pyTree):

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

# Geometry
P0 = (0,0,0); P1 = (5,0,0); P2 = (0,7,0); P3 = (5,7,0)
Ni = 20; Nj = 10
d1 = D.line(P0, P1,Ni); d2 = D.line(P2, P3,Ni)
d3 = D.line(P0, P2,Nj); d4 = D.line(P1, P3,Nj)
m = G.TFI([d1, d2, d3, d4])
C.convertPyTree2File(m, 'out.cgns')
```

Generator.**TFITri**(*a1*, *a2*, *a3*)

Generate three structured meshes by transfinite interpolation around three given curves a1, a2, a3. N3-N2+N1 must be odd.

Parameters

- **a1** (array) – first curve
- **a2** (array) – second curve
- **a3** (array) – third curve

Returns 2D structured mesh

Return type array or pyTree

Example of use:

- TFI structured mesh generation between three curves (array):

```
# - TFITri (array)
import Converter as C
import Generator as G
import Geom as D

P0 = (0,0,0); P1 = (5,0,0); P2 = (1,7,0)

# 3 curves (dont need to be lines)
d1 = D.line(P0, P1, N=11)
d2 = D.line(P1, P2, N=11)
d3 = D.line(P0, P2, N=11)
r = G.TFITri(d1, d2, d3)
C.convertArrays2File(r, 'out.plt')
```

- TFI structured mesh generation between three curves (pyTree):

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

P0 = (0,0,0); P1 = (5,0,0); P2 = (1,7,0)

# 3 curves (dont need to be lines)
d1 = D.line(P0, P1, N=11)
d2 = D.line(P1, P2, N=11)
d3 = D.line(P0, P2, N=11)
r = G.TFITri(d1, d2, d3)
C.convertPyTree2File(r, 'out.cgns')
```

Generator.**TFIO**(*a*)

Generate five meshes by transfinite interpolation around one given curves a. The number of points of a must be odd.

Parameters **a** (array) – curve

Returns 2D structured mesh (butterfly O-H topology)

Return type array or pyTree

Example of use:

- Butterfly structured mesh generation by TFI (array):

```
# - TFIO (array) -
import Converter as C
import Generator as G
import Geom as D

a = D.circle((0,0,0), 1., N=41)
r = G.TFIO(a)
C.convertArrays2File(r, 'out.plt')
```

- Butterfly structured mesh generation by TFI (pyTree):

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

a = D.circle((0,0,0), 1., N=41)
r = G.TFIO(a)
C.convertPyTree2File(r, 'out.cgns')
```

Generator.**TFIHalfO**(*a1*, *a2*)

Generate four meshes by transfinite interpolation around two given curves a1 and a2 forming a half-O. N1, the number of points of a1 and N2, the number of points of a2 must be odd.

Parameters

- **a1** (array or Zone) – first curve
- **a2** (array or Zone) – second curve

Returns 2D structured mesh (half butterfly C-H topology)

Return type array or Zone

Example of use:

- Half-Butterfly structured mesh generation by TFI (array):

```
# - TFIHalfO (array) -
import Converter as C
import Generator as G
import Geom as D

a1 = D.circle((0,0,0), 1., tetas=0, tetae=180., N=41)
a2 = D.line((-1,0,0),(1,0,0), N=21)
r = G.TFIHalfO(a1, a2)
C.convertArrays2File(r, 'out.plt')
```

- Half-Butterfly structured mesh generation by TFI (pyTree):

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

a1 = D.circle((0,0,0), 1., tetas=0, tetae=180., N=41)
a2 = D.line((-1,0,0),(1,0,0), N=21)
r = G.TFIHalfO(a1, a2)
C.convertPyTree2File(r, 'out.cgns')
```

Generator.**TFIMono**(*a1*, *a2*)

Generate one mesh by transfinite interpolation around two given curves a1 and a2 forming a half-O. N1-N2 must be even.

Parameters

- **a1** (array) – first curve
- **a2** (array) – second curve

Returns 2D structured mesh

Return type array or pyTree

Example of use:

- TFI structured mesh generation between two curves (array):

```
# - TFIMono (array) -
import Converter as C
import Generator as G
import Geom as D

a1 = D.circle((0,0,0), 1., tetas=0, tetae=180., N=41)
a2 = D.line((-1,0,0),(1,0,0), N=21)
r = G.TFIMono(a1, a2)
C.convertArrays2File(r, 'out.plt')
```

- TFI structured mesh generation between two curves (pyTree):

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

a1 = D.circle((0,0,0), 1., tetas=0, tetae=180., N=41)
a2 = D.line((-1,0,0),(1,0,0), N=21)
r = G.TFIMono(a1, a2)
C.convertPyTree2File(r, 'out.cgns')
```

Generator.**hyper2D**(*line*, *distrib*, "*C*")

Generate an hyperbolic mesh (2D) of "C" or "O" type from a from a line defined by line and from a distribution defined by distrib. The resulting mesh is nearly orthogonal.

Parameters

- **line** (array) – starting line of the hyperbolic mesh
- **distrib** (array) – distribution orthogonal to the line

Returns 2D structured mesh

Return type array or pyTree

Example of use:

- Hyperbolic structured mesh generation from a line (array):

```
# - hyper2D (array) -
import Geom as D
import Generator as G
import Converter as C

msh = D.naca(12., 5001)

# Distribution
Ni = 300; Nj = 50
distrib = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,1))
a = G.hyper2D(msh, distrib, "C")
C.convertArrays2File([a], 'out.plt')
```

- Hyperbolic structured mesh generation from a line (pyTree):

```
# - hyper2D (pyTree) -
import Geom.PyTree as D
import Generator.PyTree as G
import Converter.PyTree as C

line = D.naca(12., 5001)
# Distribution
Ni = 300; Nj = 50
distrib = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,1))

a = G.hyper2D(line, distrib, "C")
C.convertPyTree2File(a, 'out.cgns')
```

Generator.PolyLine.**polyLineMesher**(*a*, *h*, *hf*, *density*)

Generate a 2D mesh around a 2D polyline where a is the input polyline (BAR-array), h is the height of the mesh, hf is the height of the first cell and density is the number of points per unity of length. In the 'array' version, it returns a list where B[0] is the list of generated meshes, B[1] is the list of wall boundaries, B[2] is the list of overlap boundaries, B[3] is h, B[4] is density (eventually modified by the mesher). In the pyTree version, it returns a list [zones,hs,densities], where zones is a list of zones of a CGNS python tree, containing the blocks, wall boundaries, match and overlap boundaries; hs is the list of heights (modified if necessary), and densities the list of densities (also modified if necessary).

Parameters

- **a** (BAR-array) – input polyline
- **h** (float) – height of the mesh

- **hf** (float) – first cell size
- **density** (integer) – number of points per unity of length

Returns 2D structured mesh

Return type array or pyTree

Example of use:

- Structured mesh generation from a polyline (array):

```
# - polyLineMesher (array) -
import Converter as C
import Generator.PolyLine as GP
import Generator as G
import Transform as T

# Read a 2D geometry created with tecplot
a = C.convertFile2Arrays('fusee.plt')
a = G.close(a,1e-2); a = T.reorder(a,(-1,2,3))
C.convertArrays2File(a, 'input.plt')

# Data
h = 0.02; hf = 0.0001; density = 500

# Per families
coords = []; walls = []
for i in a:
    b = GP.polyLineMesher(i, h, hf, density)
    coords.append(b[0])
    walls.append(b[1])

# Flat
meshes = []
for i in coords: meshes = meshes + i

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

- Structured mesh generation from a polyline (pyTree):

```
# - polyLineMesher (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Transform.PyTree as T

tb = C.convertFile2PyTree('fusee.plt')
tb = G.close(tb,1e-2); tb = T.reorder(tb,(-1,2,3))
```

```
h = 0.02; hf = 0.0001; density = 500

res = G.polyLineMesher(tb[2][1][2][0], h, hf, density)
zones = res[0]; h = res[1]; density = res[2]
t = C.newPyTree(['PolyC1']); t[2][1][2] += zones
C.convertPyTree2File(t, 'out.cgns')
```

Generator.PolyC1.**polyC1Mesher**(*A*, *h*, *hf*, *density*, *splitCrit=10.*)

Generate a 2D mesh around a 2D polyC1 curve where A is a list of i-arrays each representing a C1 curve. All i-arrays put together must represent a polyC1 curve. SplitCrit is a curvature radius triggering split. Other arguments are similar to polyLineMesher. The function return is also similar to polyLineMesher.

Parameters

- **A** (list of arrays) – list of 1D curves
- **h** (float) – height of the mesh
- **hf** (float) – first cell size
- **density** (integer) – number of points per unity of length
- **splitCrit** (float) – threshold curvature radius below which the initial curve is split

Returns 2D structured mesh

Return type array or pyTree

Example of use:

- Structured mesh generation from a C1 line (array):

```
# - polyC1Mesher (array) -
import Converter as C
import Generator.PolyC1 as GP
import Generator as G
import Transform as T

# Read geometry from svg file
a = C.convertFile2Arrays('Data/curve.svg', density=1)[0]
a = T.homothety(a,(0,0,0),0.01)
a = T.reorder(a, (1,2,3))

h = 0.2; hp = 0.001; density = 10.; splitCrit = 2.
m = GP.polyC1Mesher(a, h, hp, density, splitCrit)

for i in m[0]:
```

```
    v = G.getVolumeMap(i)
    min = C.getMinValue(v, 'vol')
    if min <= 0:
        print('negative volume detected.')

C.convertArrays2File(m[0], 'out.plt')
```

- Structured mesh generation from a C1 line (pyTree):

```
# - polyC1Mesher (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Transform.PyTree as T

tb = C.convertFile2PyTree('curve1.svg', nptsCurve=100, nptsLine=400)
z = T.homothety(tb[2][1][2][0], (0.,0.,0.), 0.01)
z = T.reorder(z, (-1,2,3))
h = 0.1; hf = 0.001; density = 100; splitCrit = 10.
res = G.polyC1Mesher(z, h, hf, density, splitCrit)
zones = res[0]; h = res[1]; density = res[2]
t = C.newPyTree(['Base']); t[2][1][2] += zones
C.convertPyTree2File(t, 'out.cgns')
```

Generator.**pointedHat**(*a*, *(x, y, z)*)

Create a structured mesh from a curve defined by a i-array and a point. For the pyTree version: if a contains a solution, it is not taken into account in b.

Parameters

- **a** (array) – closed 1D curve
- **(x,y,z)** (3-tuple of floats) – coordinates of point

Returns 2D structured mesh

Return type array or pyTree

Example of use:

- 2D closing mesh generation from a closed curve and a point (array):

```
# - pointedHat (array) -
import Geom as D
import Generator as G
import Converter as C
c = D.circle( (0,0,0), 1., 360., 0., 100)
surf = G.pointedHat(c,(0.,0.,1.))
C.convertArrays2File([surf], 'out.plt')
```

- 2D closing mesh generation from a closed curve and a point (pyTree):

```
# - pointedHat (pyTree) -
import Geom.PyTree as D
import Generator.PyTree as G
import Converter.PyTree as C
c = D.circle( (0,0,0), 1., 360., 0., 100)
surf = G.pointedHat(c,(0.,0.,1.))
t = C.newPyTree(['Base']); t[2][1][2].append(surf)
C.convertPyTree2File(t, 'out.cgns')
```

Generator.**stitchedHat**(*a*, *(offx, offy, offz)*, *tol=1.e-6*, *tol2=1.e-5*)

Create a stitched mesh from a curve defined by a i-array. The surface is stitched in the middle. Tol is the accuracy of the search, tol2 is a merging tolerance and offx, offy, off z an optional offset. For the pyTree version: if a contains a solution, it is not taken into account in b.

Parameters

- **a** (array) – closed 1D curve
- **(offx,offy,offz)** (3-tuple of floats) – coordinates of offset vector
- **tol** (float) – accuracy of search
- **tol2** (float) – merging tolerance

Returns 2D structured mesh

Return type array or pyTree

Example of use:

- 2D stitched mesh generation from a closed curve (array):

```
# - stitchedHat (array) -
import Geom as D
import Generator as G
import Transform as T
import Converter as C

c = D.circle( (0,0,0), 1., 360., 0., 100)
c = T.contract(c, (0,0,0), (0,1,0), (0,0,1), 0.1)
c = G.stitchedHat(c, (0,0,0), 1.e-3)
C.convertArrays2File([c], 'out.plt')
```

- 2D stitched mesh generation from a closed curve (pyTree):

```
# - stitchedHat (pyTree) -
import Geom.PyTree as D
import Generator.PyTree as G
import Transform.PyTree as T
import Converter.PyTree as C
c = D.circle((0,0,0), 1., 360., 0., 100)
c = T.contract(c, (0,0,0), (0,1,0), (0,0,1), 0.1)
c = G.stitchedHat(c, (0,0,0), 1.e-4)
C.convertPyTree2File(c, 'out.cgns')
```

Generator.**surfaceWalk**(*surfaces, c, dj, constraints=[], niter=0, alphaRef=180., check=0, toldist=1.e-6*)

Surface extrusion starting from a curve, resulting into a surface mesh. dj is the distribution of points in the extrusion direction starting from c, niter the number of smoothing iterations. check=1 means that the extrusion stops at the layer before cells intersect alphaRef is the deviation angle wrt 180 degrees enabling to stop the extrusion before it crosses a sharp edge on the surface. toldist is a tolerance below which points are considered matching. Constraints can be set as 1D zones.

Parameters

- **surfaces** (`list of arrays`) – list of surfaces
- **c** (`array or pyTree`) – starting curve for the extrusion
- **dj** (`1D-array`) – distribution of points for the extrusion
- **constraints** (`list of arrays`) – 1D curves constraining the extrusion
- **niter** (`integer`) – number of smoothing iterations
- **alphaRef** (`float`) – deviation angle (in degrees) stopping the extrusion
- **check** (`integer`) – activation key for stopping the extrusion (0 or 1)
- **toldist** (`float`) – merging points tolerance

Returns 2D structured mesh

Return type array or pyTree

Example of use:

- 2D mesh extrusion from a curve and walking on a surface (array):

```
# - surfaceWalk (array)
import Converter as C
import Geom as D
import Transform as T
import Generator as G

# User definition of parametric curve
def f(t,u):
    x = t+u; y = t*t+1+u*u; z = u
    return (x,y,z)

# Array definition of geometry
a = D.surface(f)

c = D.circle((1.2,1.7,0.6), 0.1,N=100)
c = T.rotate(c, (1.2,1.7,0.6), (0,1,0), 90.)
c = T.reorder(c,(-1,2,3))
c = T.projectOrtho(c,[a])

h = G.cart((0.,0.,0.),(0.01,1,1),(30,1,1))
r = G.surfaceWalk([a], c, h, niter=100)
C.convertArrays2File([a,c,r], "out.plt")
```

- 2D mesh extrusion from a curve and walking on a surface (pyTree):

```
# - surfaceWalk (pyTree)
import Converter.PyTree as C
import Geom.PyTree  as D
import Transform.PyTree  as T
import Generator.PyTree  as G

# User definition of parametric curve
def f(t,u):
    x = t+u; y = t*t+1+u*u; z = u
    return (x,y,z)

# Array definition of geometry
a = D.surface(f)

c = D.circle((1.2,1.7,0.6), 0.1)
c = T.rotate(c, (1.2,1.7,0.6), (0,1,0), 90.)
c = T.reorder(c,(-1,2,3))
c = T.projectOrtho(c,[a])

h = G.cart((0.,0.,0.),(0.01,1,1),(15,1,1))
r = G.surfaceWalk([a], c, h, niter=100)
```

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

Generator.**collarMesh**(*s1*, *s2*, *dj*, *dk*, *niterj=100*, *niterk=100*, *ext=5*, *alphaRef=30.*, *type='union'*, *contour=[]*, *constraints1=[]*, *constraints2=[]*, *toldist=1.e-10*, *topology='overset'*)

Create a collar mesh at junction(s) between two surfaces s1 and s2 in union or difference assembly, using a distribution along the surface dj and a distribution in the normal direction to the wall dk. niterj and niterk are the number of smoothing iterations for j and k directions. ext is the extension of the collar mesh for difference assembly. type is the assembly type, and can be 'union' or 'difference'. alphaRef is the deviation angle wrt 180 degrees above which the walk is stopped. contour is the starting contour to create the collar grids, constraints1 and constraints2 are 1D zones defining the curves the collar grid must follow on surfaces s1 and s2 respectively. toldist is the matching point tolerance. Parameter 'topology' can be 'overset' or 'extruded', only useful in case of difference. Topology set to 'overset' results in two overlapping collar grids, whereas it results in a collar grid extruded from the surface grid in the other case.

Parameters

- **s1** (array or pyTree) – surface
- **s2** (array or pyTree) – surface
- **dj** (1D-array) – distribution of points along surfaces
- **dk** (1D-array) – distribution of points in the normal direction
- **niterj** (integer) – number of smoothing iterations in j direction
- **niterk** (integer) – number of smoothing iterations in k direction
- **ext** (integer) – extension of collar for difference assembly
- **alphaRef** (float) – deviation angle (in degrees) stopping the extrusion
- **type** (string) – type of the assembly (union or difference)
- **contour** (list of arrays) – starting curve for the collar creation
- **constraints1** (list of arrays) – 1D curves constraining the collar on s1 surface
- **constraints2** (list of arrays) – 1D curves constraining the collar on s2 surface
- **toldist** (float) – merging points tolerance
- **topology** (string) – choice of collar mesh topology (overset or extruded) in case of difference assembly

Returns 3D structured mesh

Return type array or pyTree

Example of use:

- 3D collar mesh between two surfaces (array):

```
# - collarMesh (array) -
import Converter as C
import Geom as D
import Transform as T
import Generator as G

s1 = D.sphere((0.,0.,0.),1,20)
s2 = T.translate(s1,(1.2,0.,0.)); s2 = T.homothety(s2,(0,0,0),0.5)
dhj = G.cart((0.,0.,0.),(1.e-2,1,1),(21,1,1))
dhk = G.cart((0.,0.,0.),(1.e-2,1,1),(11,1,1))
a = G.collarMesh(s1,s2, dhj,dhk,niterj=100,niterk=100,type='union')
C.convertArrays2File(a,"out.plt")
```

- 3D collar mesh between two surfaces (pyTree):

```
# - collarMesh (pyTree)
import Converter.PyTree as C
import Geom.PyTree as D
import Transform.PyTree as T
import Generator.PyTree as G

s1 = D.sphere((0.,0.,0.),1,20)
s2 = T.translate(s1,(1.2,0.,0.)); s2 = T.homothety(s2,(0,0,0),0.5)
dhj = G.cart((0.,0.,0.),(1.e-2,1,1),(21,1,1))
dhk = G.cart((0.,0.,0.),(1.e-2,1,1),(11,1,1))
a = G.collarMesh(s1,s2, dhj,dhk,niterj=100,niterk=100,type='union')
C.convertPyTree2File(a,"out.cgns")
```

3.3 Cartesian grid generators

Generator.**gencartmb**(*A*, *h*, *Dfar*, *nlvl*)

Simple Cartesian generator. Create a set of Cartesian grids (B) around a list of body grids (A). Those grids are patched with a ratio of 2. The user controls the number of levels, and the number of points for each level of grid. h is the spatial step on the finest level. Dfar is the maximal distance to the body. nlvl is a list that provides the number of points per level (nlvl[0]: finest grid), except for the finest level.

Parameters

- **A** (array/list of arrays or pyTree/list of pyTrees) – body grids
- **h** (float) – spatial step in the finest level
- **Dfar** (float) – maximal distance to the body A
- **nlvl** (list of integers) – list of number of points per level (except the finest one)

Returns 2D/3D structured mesh

Return type array or pyTree

Example of use:

- Generation of Cartesian mesh refined near body grids (array):

```
# - gencartmb (array) -
import Generator as G
import Converter as C

# body grid
a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 10., (50,50,30))
h = 1.e-1# Step of finest Cartesian grid
Dfar = 10.# Extension of far boundaries
nlvl = [5,5,5] # Nb of points per level, except the 4th level (automatic)
cartGrids = G.gencartmb([a], h, Dfar, nlvl)
C.convertArrays2File(cartGrids, 'out.plt')
```

- Generation of Cartesian mesh refined near body grids (pyTree):

```
# - gencartmb (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C

# body mesh
a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 10., (50,50,30))
h = 1.e-1  # Step of finest Cartesian grid
Dfar = 20. # Distance to far boundaries

# Nb of points per level:
# Here 4 levels, but last one is computed automatically
nlvl = [10,10,5] # nlvl[0]: coarse grid

t = C.newPyTree(['Bodies', 'CARTESIAN']); t[2][1][2].append(a)
zones = G.gencartmb(t[2][1], h, Dfar, nlvl)
```

```
t[2][2][2] += zones
C.convertPyTree2File(t, 'out.cgns')
```

Generator.**octree**(*surfs*, *snearList=[]*, *dfarList=[]*, *dfar=-1.*, *balancing=0*, *levelMax=1000*, *ratio=2*, *octant=None*)

Create a QUAD quadtree mesh in 2D or an HEXA octree mesh in 3D starting from a list of bodies and snears. Each parameter snear is the required spatial step of the octree near the corresponding body; the extension of the domaine can be provided by dfar, starting from the global bounding box of all surfaces defined by surfs. A list of extensions can be provided in dfarList, in order not to take into account a surface in the computation of the bounding box. It must be set to -1 for the surface that must not be taken into account. Parameter balancing=1 means that the octree is balanced, i.e. adjacent elements are at worst twice as big/small; levelMax is the maximum number of levels required. If ratio=2, then a classical octree mesh is built. If ratio=3, a 27-tree mesh is built, in which case the spacing ratio is 3 (and not 2) between two adjacent elements. Parameter balancing enables to balance the octree; balancing=0 means no balancing; balancing=1 means a classical balancing, whereas balancing=2 takes also into account elements sharing a common vertex.

Parameters

- **surfs** (`list of arrays/pyTrees`) – body grids
- **snears** (`list of floats`) – list of spatial step near the corresponding body
- **dfar** (`float`) – maximal distance to the body grids
- **balancing** (`integer`) – activation key for balanced octree (0, 1 or 2)
- **levelMax** (`integer`) – maximum number of levels
- **ratio** (`integer`) – spacing ratio between two adjacent elements

Returns 2D/3D unstructured mesh

Return type array or pyTree

Example of use:

- Generation of unstructured octree mesh refined near body grids (array):

```
# - octree (array) -
import Generator as G
import Converter as C
import Geom as D

s = D.circle((0,0,0), 1., N=100); snear = 0.01
```

```
res = G.octree([s], [snear], dfar=5., balancing=2)
C.convertArrays2File([res], 'out.plt')
```

- Generation of unstructured octree mesh refined near body grids (pyTree):

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

s = D.circle((0,0,0), 1., N=100); snear = 0.1
res = G.octree([s], [snear], dfar=5.)
C.convertPyTree2File(res, 'out.cgns')
```

Generator.**octree2Struct**(*octree*, *vmin=15*, *ext=0*, *optimized=1*, *merged=1*, *AMR=0*, *sizeMax=1000000*)

Convert an octree or a quadtree mesh into a set of Cartesian grids. Parameter ext is the extension of Cartesian grids in all the directions; vmin can be an integer defining the number of points in each Cartesian grid, or a list of integers, defining the number of points per refinement level. In that case, the first element of the list of vmin defines the finest level. Specifying all the levels is not mandatory. If optimized=1, the ext value is reduced by -1 at overlap borders for the coarsest grid for minimum overlapping. If merged=1, Cartesian grids are merged in order to reduce the number of created grids. If AMR=1, a set of AMR zones are generated. Parameter sizeMax can be used when merging is applied: in that case, the number of points per grid does not exceed sizeMax. Warning: to obtain multigrid blocks, optimized must be set to 0.

Parameters

- **octree** (array or pyTree) – input unstructured octree grid
- **vmin** (integer or list of integers) – number of points in all Cartesian grids or list of number of points for each octree level
- **ext** (integer) – extension of Cartesian grids (0 = no extension, N = extension of N cells in all direction)
- **optimized** (integer) – activation key for optimization of coarsest grid (0 or 1)
- **merged** (integer) – activation key for automatic merging of Cartesian grids
- **AMR** (integer) – activation key for AMR generation (0 or 1)

- **sizeMax** (integer) – maximum number of points in Cartesian grids after merging

Returns 2D/3D structured mesh

Return type array or pyTree

Example of use:

- Generation of structured Cartesian mesh from an octree grid (array):

```
# - octree2Struct (array) -
import Generator as G
import Converter as C
import Geom as D

s = D.circle((0,0,0), 1., N=100); snear = 0.1
res = G.octree([s],[snear], dfar=5., balancing=1)
res = G.octree2Struct(res, vmin=5, ext=2, optimized=1)
C.convertArrays2File([s]+res, "out.plt")
```

- Generation of structured Cartesian mesh from an octree grid (pyTree):

```
# - octree2Struct (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C
import Geom.PyTree as D

s = D.circle((0,0,0), 1., N=100); snear = 0.1
res = G.octree([s],[snear], dfar=5., balancing=1)
res = G.octree2Struct(res, vmin=5, ext=2,merged=1)
C.convertPyTree2File(res, 'out.cgns')
```

Generator.**adaptOctree**(*octree, indicator, balancing=1, ratio=2*)

Adapt an unstructured octree with respect to an indicator field located at element centers. If 'indicator' is strictly positive for an element, then the element must be refined as many times as required by the indicator number. If 'indicator' is strictly negative, the element is coarsened if possible as many times as required by the indicator number. If 'indicator' is 0., the element remains unchanged. balancing=1 means that the octree is balanced after adaptation. If ratio=2, then a classical octree mesh is built. If ratio=3, a 27-tree mesh is built, in which case the spacing ratio is 3 (and not 2) between two adjacent elements. For array interface indicator is an array, for pyTree version, indicator is the name of field stored as a solution located at centers. Exists also as in place version (_adaptOctree) that modifies a and returns None.

Parameters

- **octree** (array or pyTree) – input unstructured octree grid
- **indicator** (array or variable name in the pyTree) – field of values to indicate where to refine, coarsen or maintain the octree grid
- **balancing** (integer) – activation key for balanced octree (0, 1 or 2, see the definition of octree function for the meaning)
- **ratio** (integer) – spacing ratio between two adjacent elements

Returns modified reference copy of t

Return type same as input

Example of use:

- Adaptation of an octree grid wrt. indicator (array):

```
# - adaptOctree (array) -
import Generator as G
import Converter as C
import Geom as D

s = D.circle((0,0,0), 1., N=100); snear = 0.1
o = G.octree([s], [snear], dfar=5., balancing=1)
indic = C.node2Center(o)
indic = C.initVars(indic, 'indicator', 1.)
res = G.adaptOctree(o, indic)
C.convertArrays2File([o,res], "out.plt")
```

- Adaptation of an octree grid wrt. indicator (pyTree):

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

s = D.circle((0,0,0), 1., N=100); snear = 0.1
o = G.octree([s], [snear], dfar=5.,balancing=1)
o = C.initVars(o, 'centers:indicator', 1.)
res = G.adaptOctree(o)
C.convertPyTree2File(res, 'out.cgns')
```

Generator.**expandLayer**(*octree, level=0, corners=0, balancing=0*)

Expand the layer of given level for an octree unstructured mesh. If corners=1, expand also in corners directions. Exists also as in place version (_expandLayer) that modifies a and returns None.

Parameters

- **octree** (array or pyTree) – input unstructured octree grid
- **level** (integer) – level to be expanded (level=0 is the finest)
- **corners** (integer) – activation key for expansion in corners (0 or 1)
- **balancing** (integer) – activation key for balanced octree (0, 1 or 2, see the definition of octree function for the meaning)

Returns modified reference copy of t

Return type same as input

Example of use:

- Expansion of user-specified level in an octree grid (array):

```
# - expandLayer (array) -
import Generator as G
import Converter as C
import Geom as D

s = D.circle((0.,0.,0.), 1., N=100)
o = G.octree([s], [0.1], dfar=1., balancing=1)
o2 = G.expandLayer(o, level=0)
C.convertArrays2File([o, o2], "out.plt")
```

- Expansion of user-specified level in an octree grid (pyTree):

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

s = D.circle((0.,0.,0.),1.,N=100)
o = G.octree([s], [0.1], dfar=1., balancing=1)
o2 = G.expandLayer(o, level=0)
C.convertPyTree2File(o2, 'out.cgns')
```

3.4 Operations on meshes

Generator.**close**(*a*, *tol=1.e-12*)

Close a mesh defined by array a. Points that are distant of tol maximum to one another are merged.

Exists also as in place version (_close) that modifies a and returns None.

Parameters

- **a** (array or pyTree) – input mesh
- **tol** (float) – merging points tolerance

Returns modified reference copy of t

Return type array or pyTree

Example of use:

- Mesh closing (array):

```
# - close (array) -
import Converter as C
import Generator as G

a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0.01, 10., (20,20,10))
a = C.convertArray2Tetra(a)
a = G.close(a, 1.e-3)
C.convertArrays2File(a, 'out.plt')
```

- Mesh closing (pyTree):

```
# - close (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G

a1 = G.cart((0,0,0), (1,1,1), (10,10,1))
a2 = G.cart((9+1.e-2,0,0), (1,1,1), (10,10,1))
a3 = G.cart((0,-5.01,0),(1,1,1),(19,6,1))
a4 = G.cart((0,9.0001,0),(1,1,1),(10,6,1))
a5 = G.cart((9.01,9.0002,0),(1,1,1),(10,6,1))
t = C.newPyTree(['Base',2,a1,a2,a3,a4,a5])
t = G.close(t, 1.e-1)
C.convertPyTree2File(t, 'out.cgns')
```

Generator.**selectInsideElts**(*a*, *curves*)

Select elements of a TRI-array, whose centers are inside the given list of curves, defined by BAR-arrays.

Parameters

- **a** (array or pyTree) – input triangle 2D mesh
- **curves** (array or list of arrays) – list of curves

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Selection of TRI cells inside a specified curve (array):

```
# - selectInsideElts (array) -
import Converter as C
import Generator as G
import Geom as D

a = G.cart( (0,0,0), (1,1,1), (10,10,1)); a = C.convertArray2Tetra(a)
b = D.circle( (5,5,0), 3.); b = C.convertArray2Tetra(b)
a = G.selectInsideElts(a, [b])
C.convertArrays2File([a,b], 'out.plt')
```

- Selection of TRI cells inside a specified curve (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (10,10,1))
a = C.convertArray2Tetra(a)
b = D.circle((5,5,0), 3.)
b = C.convertArray2Tetra(b)
a = G.selectInsideElts(a, b)
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**map**(*a*, *distrib*, *dir*)

Map a distribution on a curve or on a structured surface. Map a i-array distribution in a direction (dir=1,2,3) in a surface or volume mesh.

Parameters

- **a** (array or pyTree) – 1D/2D/3D structured mesh
- **distrib** (array) – distribution of points
- **dir** (integer) – direction i/j/k for the distribution (dir=1,2,3)

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Map distribution (array):

```
# - map (array) -
import Geom as D
import Generator as G
import Converter as C

# Map on a curve
l = D.line( (0,0,0), (1,1,0) )
Ni = 10
d = G.cart( (0,0,0), (1./(Ni-1),1.,1.), (Ni,1,1) )
m = G.map(l, d)
C.convertArrays2File([m], "out1.plt")

# Map on a structured surface
ni = 2; nj = 3
a = G.cart((0,0,0), (1,1,1), (ni,nj,1))
C.setValue(a, (1,1,1), [1.,1.,2.])
C.setValue(a, (1,2,1), [1.,2.,5.])
C.setValue(a, (1,3,1), [1.,3.,2.])
C.setValue(a, (2,1,1), [2.,1.,2.])
C.setValue(a, (2,2,1), [2.,2.,5.])
C.setValue(a, (2,3,1), [2.,3.,2.])
b = D.bezier(a, 10, 10)
Ni = 50; Nj = 30
d = G.cart( (0,0,0), (1./(Ni-1),1./(Nj-1),1.), (Ni,Nj,1) )
d = G.enforceX(d, 0.5, 0.01, (10,20))
d = G.enforceY(d, 0.5, 0.01, (10,20))
b = G.map(b, d)
C.convertArrays2File([b], "out2.plt")

# Map in a direction
a = G.cylinder((0,0,0), 0.5, 2., 0, 60, 1., (20,20,1))
Ni = 10
d = G.cart( (0,0,0), (1./(Ni-1),1.,1.), (Ni,1,1) )
d = G.enforcePlusX(d, 0.01, (10,20))
a = G.map(a, d, 2)
C.convertArrays2File([a], "out3.plt")
```

- Map distribution (pyTree):

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

l = D.line( (0,0,0), (1,1,0) )
```

```
Ni = 11; dist = G.cart( (0,0,0), (1./(Ni-1),1.,1.), (Ni,1,1) )
l = G.map(l, dist)
t = C.newPyTree(['Base',1]); t[2][1][2].append(l)
C.convertPyTree2File(t, 'out.cgns')
```

Generator.**mapSplit**(*a*, *distrib*, *splitCrit=100.*)

Split a i-array and map a distribution on the splitted i-array. SplitCrit is the curvature radius triggering split.

Parameters

- **a** (array or pyTree) – 1D/2D/3D structured mesh
- **distrib** (array) – distribution of points
- **splitCrit** (float) – curvature radius for array splitting

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Split and map distribution (array):

```
# - mapSplit (array) -
import Generator as G
import Converter as C
import Geom as D

# polyline
a = D.polyline([(0,0,0),(1,0,0),(1,1,0),(2,3,0),(1.5,3,0),(1,1.5,0),(0,0,0)])
# distribution
Ni = 41
dist = G.cart((0,0,0),(1./(Ni-1),1,1),(Ni,1,1))
dist = G.enforceX(dist, 15.5/(Ni-1), 0.005, 2,5)
dist = G.enforceX(dist, 27.5/(Ni-1), 0.005, 2,5)
a = G.mapSplit(a,dist,0.25)
C.convertArrays2File(a, 'out.plt')
```

- Split and map distribution (pyTree):

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

```
# polyline
a = D.polyline([(0,0,0),(1,0,0),(1,1,0),(2,3,0),(1.5,3,0),(1,1.5,0),(0,0,0)])
# distribution
Ni = 41
dist = G.cart((0,0,0),(1./(Ni-1),1,1),(Ni,1,1))
dist = G.enforceX(dist, 15.5/(Ni-1), 0.005, 2,5)
dist = G.enforceX(dist, 27.5/(Ni-1), 0.005, 2,5)
zones = G.mapSplit(a,dist,0.25)
t = C.newPyTree(['Base',1]); t[2][1][2] += zones
C.convertPyTree2File(t, 'out.cgns')
```

Generator.**refine**(*a*, *power*, *dir*)

Refine a structured array. The original distribution is kept but the number of points is multiplied by power. Dir is the direction of refinement (1, 2, 3). If dir=0, refine in all directions.

Exists also as in place version (_refine) that modifies a and returns None.

Parameters

- **a** ([array] or [zone]) – 1D/2D/3D structured mesh
- **power** (float) – multiplication factor of number of points
- **dir** (integer) – direction i/j/k for the distribution (dir=0,1,2,3)

Return type Identical to a

Example of use:

- Structured mesh refinement (array):

```
# - refine (array) -
import Generator as G
import Converter as C

a = G.cart( (0,0,0), (0.1,0.1,0.1), (20,20,1) )

a = G.refine(a, 1.5, 1)
C.convertArrays2File([a], 'out.plt')
```

- Structured mesh refinement (pyTree):

```
# - refine (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C

a = G.cart( (0,0,0), (0.1,0.1,0.1), (20,20,1) )
```

```
a = G.refine(a, 1.5, 1)
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**mapCurvature**(*a*, *N*, *power*, *dir*)

Map a structured array following the curvature. N is the final number of points. Dir is the direction of remeshing (1, 2, 3).

Parameters

- **a** ([array] or [zone]) – 1D/2D structured mesh
- **N** (integer) – number of points after new distribution
- **power** (float) – refinement factor
- **dir** (integer) – direction i/j/k for the distribution (dir=1,2,3)

Return type Identical to a

Example of use:

- Map distribution wrt. curvature (array):

```
# - mapCurvature (array) -
import Generator as G
import Converter as C
import Geom as D

ni = 2; nj = 3
a = G.cart((0,0,0), (1,1,1), (ni,nj,1))
C.setValue(a, (1,1,1), [1.,1.,2.])
C.setValue(a, (1,2,1), [1.,2.,4.])
C.setValue(a, (1,3,1), [1.,3.,2.])
C.setValue(a, (2,1,1), [2.,1.,2.])
C.setValue(a, (2,2,1), [2.,2.,5.])
C.setValue(a, (2,3,1), [2.,3.,2.])
b = D.bezier(a, density=10.)

b = G.mapCurvature(b, N=100, power=0.5, dir=1)
C.convertArrays2File([b], 'out.plt')
```

- Map distribution wrt. curvature (pyTree):

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

```
ni = 2; nj = 3
a = G.cart((0,0,0), (1,1,1), (ni,nj,1))
C.setValue(a,'GridCoordinates', (1,1,1), [1.,1.,2.])
C.setValue(a,'GridCoordinates', (1,2,1), [1.,2.,5.])
C.setValue(a,'GridCoordinates', (1,3,1), [1.,3.,2.])
C.setValue(a,'GridCoordinates', (2,1,1), [2.,1.,2.])
C.setValue(a,'GridCoordinates', (2,2,1), [2.,2.,5.])
C.setValue(a,'GridCoordinates', (2,3,1), [2.,3.,2.])
b = D.bezier(a, density=10.)

b = G.mapCurvature(b, N=100, power=0.5, dir=1)

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

Generator.**densify**(*a*, *h*)

Densify a i-array or a BAR-array with a new discretization step h. Discretization points from the original array are kept.

Exists also as in place version (_densify) that modifies a and returns None.

Parameters

- **a** (`[array] or [zone]`) – 1D structured mesh
- **h** (`float`) – new cell size step for the points densification

Return type Identical to a

Example of use:

- Curve densification (array):

```
# - densify (array) -
import Generator as G
import Converter as C
import Geom as D

a = D.circle((0,0,0), 1., 10)
b = G.densify(a, 0.01)
C.convertArrays2File(b, 'out.plt')
```

- Curve densification (pyTree):

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

```
a = D.circle((0,0,0), 1., 10)
b = G.densify(a, 0.01)
C.convertPyTree2File(b, 'out.cgns')
```

Generator.**grow**(*a*, *vector*)

Grow a surface array of one layer. Vector is the node displacement. For the array version, vector is defined by an array. For the PyTree version, vector = ['v1','v2','v3'] where variables 'v1', 'v2', 'v3' are defined as solutions in a, located at nodes.

Parameters

- **a** (array or Zone) – 2D surface mesh
- **vector** (array or list of 3 variables contained in the solution) – vector of node displacement

Returns new 3D structured mesh

Return type array or Zone

Example of use:

- Extrusion of one layer from a surface mesh (array):

```
# - grow (array) -
import Converter as C
import Generator as G
import Geom as D

a = D.sphere( (0,0,0), 1., 50 )
n = G.getNormalMap(a)
n = C.center2Node(n); n[1] = n[1]*100.
b = G.grow(a, n)
C.convertArrays2File([b], 'out.plt')
```

- Extrusion of one layer from a surface mesh (pyTree):

```
# - grow (pyTree)-
import Converter.PyTree as C
import Converter.Internal as Internal
import Generator.PyTree as G
import Geom.PyTree as D

a = D.sphere((0,0,0), 1., 50)
a = G.getNormalMap(a)
a = C.center2Node(a, Internal.__FlowSolutionCenters__)
```

```
a = C.rmVars(a, Internal.__FlowSolutionCenters__)
b = G.grow(a, ['sx','sy','sz'])
t = C.newPyTree(['Base1',2,'Base2',3])
t[2][1][2].append(a); t[2][2][2].append(b)
C.convertPyTree2File(t, 'out.cgns')
```

Generator.**stack**(*a*, *b=None*)

Stack two 2D structured meshes or a list of structured meshes (with the same nixnj) into a single 3D mesh.

Parameters

- **a** (array, Zone or list of arrays, zones) – a 2D structured mesh or a list of structured meshes
- **b** (array or Zone) – 2D structured mesh or None

Returns new 3D structured mesh

Return type array or Zone

Example of use:

- Mesh generation by stacking two meshes (array):

```
# - stack (array) -
import Generator as G
import Converter as C
import Transform as T
import Geom as D

# Concatenate 2 structured grids
a = G.cylinder((0,0,0), 1, 1.3, 360, 0, 1., (50,10,1))
b = T.rotate(a, (0,0,0), (1,0,0), 5.)
b = T.translate(b, (0,0,0.5))
c = G.stack(a, b)

# Concatenate a list of structured grids
a = []
for i in range(10):
    a.append(D.circle((0,0,i), 1.))
c = G.stack(a)

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

- Mesh generation by stacking two meshes (pyTree):

```
# - stack (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C
import Transform.PyTree as T

a = G.cylinder((0,0,0), 1, 1.3, 360, 0, 1., (50,10,1))
b = T.rotate(a, (0,0,0), (1,0,0), 5.)
b = T.translate(b, (0,0,0.5))
c = G.stack(a, b)
C.convertPyTree2File(c, 'out.cgns')
```

Generator.**addNormalLayers**(*a*, *d*, *check=0*, *niter=0*)

Normal extrusion from a surface mesh. d is a 1D distribution providing the height of each layer. If check=1, the extrusion stops before negative volume cells are created. Niter specifies the number of iterations for normals smoothing.

Parameters

- **a** (array or pyTree) – 2D surface mesh
- **d** (1D array) – distribution of normal extrusion
- **check** (integer) – activation key for negative volume criteria (0 or 1)
- **niter** (integer) – number of iterations for normals smoothing

Returns new 3D structured mesh

Return type array or pyTree

Example of use:

- Normal extrusion from a surface mesh (array):

```
# - addNormalLayers (array) -
import Generator as G
import Converter as C
import Geom as D

d = C.array('d', 3, 1, 1)
d[1][0,0] = 0.1; d[1][0,1] = 0.2; d[1][0,2] = 0.3
a = D.sphere( (0,0,0), 1, 50 )
a = G.addNormalLayers(a, d)
C.convertArrays2File([a], 'out.plt')
```

- Normal extrusion from a surface mesh (pyTree):

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

d = G.cart((0.1,0.,0.), (0.1,1,1),(2,1,1))
a = D.sphere((0,0,0), 1, 50)
a = D.line((0,0,0),(1,1,0),3)
b = G.addNormalLayers(a, d)
d = G.cart((0.1,0.,0.), (-0.1,1,1),(2,1,1))
c =  G.addNormalLayers(a, d)
import Transform.PyTree as T
a = T.join(b,c)
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**TTM**(*a*, *niter=100*)

Smooth a mesh using elliptic generator.

Parameters

- **a** (array or pyTree) – 2D structured mesh
- **niter** (integer) – number of smoothing iterations

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- 2D structured mesh smoothing (array):

```
# - TTM (array) -
import Converter as C
import Generator as G
import Geom as D

P0 = (0,0,0); P1 = (5,0,0); P2 = (0,7,0); P3 = (5,7,0)

# Geometry
d1 = D.line(P0, P1); d2 = D.line(P2, P3)
pts = C.array('x,y,z', 5, 1, 1)
x = pts[1][0]; y = pts[1][1]; z = pts[1][2]

x[0] = 0. ; y[0] = 0.; z[0] = 0.
x[1] =-2. ; y[ 1 ] = 2.; z[1] = 0.
x[2] =-3. ; y[ 2 ] = 3.; z[2] = 0.
x[3] = 2. ; y[ 3 ] = 5.; z[3] = 0.
```

```
x[4] = 0. ; y[ 4 ] = 7.; z[4] = 0.
b1 = D.bezier(pts)

x[0] = 5.; y[ 0 ] = 0.; z[ 0 ] = 0.
x[1] = 3.; y[ 1 ] = 2.; z[ 1 ] = 0.
x[2] = 2.; y[ 2 ] = 3.; z[ 2 ] = 0.
x[3] = 6.; y[ 3 ] = 5.; z[ 3 ] = 0.
x[4] = 5.; y[ 4 ] = 7.; z[ 4 ] = 0.
b2 = D.bezier( pts )
C.convertArrays2File([d1, d2, b1, b2], 'geom.plt')

# Regular discretision of each line
Ni = 20; Nj = 10
r = G.cart((0,0,0), (1./(Ni-1),1,1), (Ni,1,1))
q = G.cart((0,0,0), (1./(Nj-1),1,1), (Nj,1,1))
r1 = G.map(d1, r)
r2 = G.map(d2, r)
r3 = G.map(b1, q)
r4 = G.map(b2, q)

# TTM
m = G.TFI([r1, r2, r3, r4])
m2 = G.TTM(m, 2000)
C.convertArrays2File([m,m2], 'out.plt')
```

- 2D structured mesh smoothing (pyTree):

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

P0 = (0,0,0); P1 = (5,0,0); P2 = (0,7,0); P3 = (5,7,0)

# Geometry
d1 = D.line(P0,P1); d2 = D.line(P2,P3)
d3 = D.line(P0,P2); d4 = D.line(P1,P3)

# Regular discretisation of each line
Ni = 20; Nj = 10
r = G.cart((0,0,0), (1./(Ni-1),1,1), (Ni,1,1))
q = G.cart((0,0,0), (1./(Nj-1),1,1), (Nj,1,1))
r1 = G.map(d1, r); r2 = G.map(d2, r)
r3 = G.map(d3, q); r4 = G.map(d4, q)

# TTM
```

```
m = G.TFI([r1, r2, r3, r4])
m = G.TTM(m)
t = C.newPyTree(['Base',2]); t[2][1][2].append(m)
C.convertPyTree2File(t, 'out.cgns')
```

Generator.**snapFront**(*a*, *S*, *optimized=1*)

Snap a mesh to a surface S. A front must be defined in a by a cellN field. Points of this front are snapped to the surface. If optimized=0, the exterior front cellN=1 is snapped, else if optimized=1 optimized front cellN=1 is snapped, else if optimized=2, front cellN=0 is snapped.

Exists also as in place version (_snapFront) that modifies a and returns None.

Parameters

- **a** (array or pyTree) – 3D mesh
- **S** (list of zones) – surface mesh
- **optimized** (integer) – optimization key (0,1,2)

Returns new unstructured mesh

Return type array or pyTree

Example of use:

- Mesh snapping to a surface (array):

```
# - snapFront (array) -
import Generator as G
import Converter as C
import Geom as D
import Connector as X
import Transform as T

s = D.circle((0,0,0), 1., N=100)
s = T.addkplane(s)

# Grille cartesienne (reguliere)
BB = G.bbox([s])
ni = 100; nj = 100; nk = 3
xmin = BB[0]; ymin = BB[1]; zmin = BB[2]-0.5
xmax = BB[3]; ymax = BB[4]; zmax = BB[5]+0.5
hi = (xmax-xmin)/(ni-1); hj = (ymax-ymin)/(nj-1)
h = min(hi, hj)
ni = int((xmax-xmin)/h)+7; nj = int((ymax-ymin)/h)+7
b = G.cart((xmin-3*h, ymin-3*h, zmin), (h, h, 1.), (ni,nj,nk))
celln = C.array('cellN', ni, nj, nk)
```

```
celln = C.initVars(celln, 'cellN', 1.)

# Masquage
cellno = X.blankCells([b], [celln], [s], blankingType=0, delta=0., dim=2)
a = C.initVars(s, 'cellN', 1)
b = C.addVars([b, cellno[0]])

# Adapte le front de la grille a la surface
b = T.subzone(b, (1,1,2), (b[2],b[3],2))
b = G.snapFront(b, [s])

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

- Mesh snapping to a surface (pyTree):

```
# - snapFront (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C
import Geom.PyTree as D
import Connector.PyTree as X
import Transform.PyTree as T
import Converter.Internal as Internal

s = D.circle((0,0,0), 1., N=100)
s2 = T.addkplane(s)

# Grille cartesienne (reguliere)
BB = G.bbox([s])
ni = 100; nj = 100; nk = 3
xmin = BB[0]; ymin = BB[1]; zmin = BB[2]-0.5
xmax = BB[3]; ymax = BB[4]; zmax = BB[5]+0.5
hi = (xmax-xmin)/(ni-1); hj = (ymax-ymin)/(nj-1)
h = min(hi, hj)
ni = int((xmax-xmin)/h)+7; nj = int((ymax-ymin)/h)+7
b = G.cart( (xmin-3*h, ymin-3*h, zmin), (h, h, 1.), (ni,nj,nk) )
t = C.newPyTree(['Cart'])
t[2][1][2].append(b)

# Masquage
t = C.initVars(t,'cellN',1)
import numpy
BM = numpy.array([[1]])
t = X.blankCells(t,[[s2]],BM,blankingType='node_in',dim=2)

# Adapte le front de la grille a la surface
dim = Internal.getZoneDim(b)
```

```
t = T.subzone(t, (1,1,2), (dim[1],dim[2],2))
t = G.snapFront(t, [s2])

t = C.addBase2PyTree(t, 'Surface', cellDim=2)
s2 = C.initVars(s2,'cellN',1)
t[2][2][2].append(s2)

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

Generator.**snapSharpEdges**(*a*, *S*, *step=None*, *angle=30.*)

Snap a mesh to a surface S, constrained by sharp edges and corners. if step != None, sharp edges are refined with this step. Sharp Edges are calculated depending on angle.

Exists also as in place version (_snapSharpEdges) that modifies a and returns None.

Parameters

- **a** (array or pyTree) – mesh to be deformed
- **S** (list of zones) – surface mesh
- **step** (float) – step for sharp edges refinement
- **angle** (float) – angle (in degrees) for sharp edges detection

Returns new unstructured mesh

Return type array or pyTree

Example of use:

- Mesh snapping to sharp edges of a surface (array):

```
# - snapSharpEdges (array) -
import Generator as G
import Converter as C
import Post as P
import Geom as D

# Enforce polyline define by s in b
s = D.polyline([(0.2,0,0),(1,1,0),(2.5,1,0),(0.2,0,0)])
s = C.initVars(s, 'indic', 0)
h = 0.1
ni = 30; nj = 20; nk=1
b = G.cartHexa((-0.5, -0.5, 0), (h, h, 1.), (ni,nj,nk))
```

```
b = C.initVars(b, 'indic', 0)
b = G.snapSharpEdges(b, [s], h)
c = C.converter.convertQuad2Tri(b)
C.convertArrays2File([b,c, s], 'out.plt')

# Same with smooth
#c = T.smooth(c, eps=0.5, niter=5,
#             fixedConstraints=s)
#            #projConstraints=s)C

# Enforce all constraints (must be over-refined)
s = D.circle((0,0,0), R=1, N=400)
s = C.initVars(s, 'indic', 0)
h = 0.3
ni = 10; nj = 20; nk=1
b = G.cartHexa((-1.5, -1.5, 0), (h, h, 1.), (ni,nj,nk))
b = C.initVars(b, 'indic', 0)
b = G.snapSharpEdges(b, [s], 0.1*h)
c = C.converter.convertQuad2Tri(b)
C.convertArrays2File([b,c,s], 'out.plt')
import sys; sys.exit()

# Idem external
h = 0.3
ni = 10; nj = 10; nk=1
s = G.cartHexa((-1.6,-1.6,0), (h/2,h/2,1.), (2*ni+2,2*nj+2,nk))
s = P.exteriorFaces(s)
s = C.initVars(s, 'indic', 0)
b = G.cartHexa((-1.5, -1.5, 0), (h, h, 1.), (ni,nj,nk))
b = C.initVars(b, 'indic', 0)
b = G.snapSharpEdges(b, [s], h*0.1)
C.convertArrays2File([b,s], 'out.plt')
```

- Mesh snapping to sharp edges of a surface (pyTree):

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

# polylignes avec angles vifs
s = D.polyline([(0.02,0,0),(1,1,0),(2,1,0),(0.02,0,0)])
# Grille cartesienne (reguliere)
h = 0.1
ni = 30; nj = 20; nk=1
b = G.cart((-0.5, -0.5, 0), (h, h, 1.), (ni,nj,nk))
```

```
b = G.snapSharpEdges(b, [s], h)
t = C.newPyTree(['Cart','Surface'])
t[2][1][2].append(b); t[2][2][2].append(s)
C.convertPyTree2File(t, 'out.cgns')
```

3.5 Operation on surface meshes

Generator.**fittingPlaster**(*a*, *bumpFactor=0.*)

Fit a surface structured patch to a curve a. BumpFactor controls the curvature of the patch.

Parameters

- **a** (array) – curve to deform the structured patch
- **bumpFactor** (float) – amplitude of the bump

Returns new structured mesh

Return type array or pyTree

Example of use:

- Cartesian mesh deformation by a curve (array):

```
# - fittingPlaster (array) -
import Generator as G
import Converter as C
import Geom as D

a = D.circle( (0,0,0), 1, N=50 )
a = C.convertArray2Tetra(a)
a = G.close(a)
b = G.fittingPlaster(a, bumpFactor=0.5)
C.convertArrays2File([a,b], 'out.plt')
```

- Cartesian mesh deformation by a curve (pyTree):

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

a = D.circle( (0,0,0), 1, N=50 )
a = C.convertArray2Tetra(a)
a = G.close(a)
```

```
b = G.fittingPlaster(a, bumpFactor=0.5)
C.convertPyTree2File(b, 'out.cgns')
```

Generator.**gapfixer**(*a*, *c*, *hardPoints=None*, *refine=1*)

Fill a gap defined by a BAR contour a drawn on a surface c. You can force the generated mesh to pass through hardPoints (NODES). If refine=0, no inside points are added.

Parameters

- **a** (BAR array) – contour of the gap
- **c** (array) – surface to be filled
- **hardPoints** (array or list of arrays) – mesh containing nodes to be enforced
- **refine** (integer) – activation key for including points in the gap mesh (0 or 1)

Returns new surface mesh

Return type array or pyTree

Example of use:

- Surface gap filling from a contour (array):

```
# - gapfixer (array) -
import Generator as G
import Converter as C
import Geom as D

# Fix the gap inside a circle drawn on a plane
a = D.circle((0,0,0), 1, N=100)
a = C.convertArray2Tetra(a); a = G.close(a)
b = G.cart((-2.,-2.,0.), (0.1,0.1,1.), (50,50,1))
a1 = G.gapfixer(a, b)
C.convertArrays2File(a1, 'out.plt')

# Fill the gap in the circle, using one defined point
hp = D.point((0.5, 0.5, 0.))
a2 = G.gapfixer(a, b, hp, refine=0)
C.convertArrays2File(a2, 'outHP.plt')
```

- Surface gap filling from a contour (pyTree):

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

# Fix the gap inside a circle drawn on a plane
a = D.circle((0,0,0), 1, N=100)
a = C.convertArray2Tetra(a); a = G.close(a)
b = G.cart((-2.,-2.,0.), (0.1,0.1,1.), (50,50,1))
a1 = G.gapfixer(a, b)
C.convertPyTree2File(a1, 'out.cgns')

# Fill the gap in the circle, using one defined point
hp = D.point((0.5, 0.5, 0.))
a2 = G.gapfixer(a, b, hp, refine=0)
C.convertPyTree2File(a2, 'outHP.cgns')
```

Generator.**gapsmanager**(*A*, *mode=0*, *coplanar=0*)

Fill multiple gaps in a set of surface components A. Also, eliminate overlap regions between components if any. Normals for all patches must be pointed outwards. Set mode=0 for nodal mesh, 1 for center mesh, and 2 otherwise. Set coplanar=1 if all components are lying on a same plane.

Parameters

- **A** (array or pyTree) – surface mesh with gaps
- **mode** (integer) – key for grid location (0 = nodes, 1 = centers, 2 = others)
- **coplanar** (integer) – activation key for coplanar components of A (0 or 1)

Returns new surface mesh

Return type array or pyTree

Example of use:

- Surface gaps filling (array):

```
# - gapsmanager (array) -
import Geom as D
import Converter as C
import Generator as G

a = D.sphere6((0,0,0), 1, N=10)
a = C.node2Center(a)
```

```
a = C.convertArray2Tetra(a)
b = G.gapsmanager(a, mode=2)
C.convertArrays2File(b, 'out.plt')
```

- Surface gaps filling (pyTree):

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

a = D.sphere6((0,0,0), 1, N=10)
a = C.node2Center(a)
a = C.convertArray2Tetra(a)
b = G.gapsmanager(a, mode=2)
C.convertPyTree2File(b, 'out.cgns')
```

3.6 Information on generated meshes

Generator.**barycenter**(*a*, *weight=None*)

Return the barycenter of a, with optional weight.

Parameters

- **a** (array or pyTree) – input mesh
- **weight** (string) – name of the weight variable in a

Returns coordinates of the barycenter

Return type 3-list of floats

Example of use:

- Computation of the mesh barycenter (array):

```
# - barycenter (array) -
import Generator as G
import Converter as C
a = G.cart((0.,0.,0.), (0.1,0.1,1.), (20,20,20))
print(G.barycenter(a))
w = C.initVars(a, 'weight', 1.); w = C.extractVars(w,['weight'])
print(G.barycenter(a, w))
```

- Computation of the mesh barycenter (pyTree):

```
# - barycenter (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C
a = G.cart((0.,0.,0.), (0.1,0.1,1.), (20,20,20))
print(G.barycenter(a))
a = C.initVars(a, 'weight', 1.)
print(G.barycenter(a, 'weight'))
```

Generator.**bbox**(*a*)

Return the bounding box [xmin, ymin, zmin, xmax, ymax, zmax] of a.

Parameters **a** (array or pyTree) – input mesh

Returns coordinates of the bounding box

Return type 6-list of floats

Example of use:

- Computation of the mesh bounding box (array):

```
# - bbox (array) -
import Generator as G
a = G.cart((0.,0.,0.),(0.1,0.1,1.),(20,20,20))
b = G.cart((12.,0.,0.),(0.1,0.1,1.),(20,20,20))
print(G.bbox(a))
print(G.bbox([a, b]))
```

- Computation of the mesh bounding box (pyTree):

```
# - bbox (pyTree) -
import Generator.PyTree as G
a = G.cart((0.,0.,0.),(0.1,0.1,1.),(20,20,20))
print(G.bbox(a))
```

Generator.**bboxOfCells**(*a*)

Return the bounding box of each cell of a. The bounding box field is located at centers of cells.

Exists also as in place version (_bboxOfCells) that modifies a and returns None.

Parameters **a** (array or pyTree) – input mesh

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Computation of the cell bounding boxes (array):

```
# - bboxOfCells (array) -
import Generator as G
a = G.cart((0.,0.,0.),(0.1,0.1,1.),(20,20,20))
b = G.bboxOfCells(a)
print(b)
```

- Computation of the cell bounding boxes (pyTree):

```
# - bboxOfCells (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C
a = G.cart((0.,0.,0.),(0.1,0.1,1.),(20,20,20))
a = G.bboxOfCells(a)
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**BB**(*a*, *method='AABB'*, *weighting=0*)

Return the bounding box of a as an array or a zone. If method is 'AABB', then it computes the Axis-Aligned Bounding-Box, if method is 'OBB' then it computes the Oriented Bounding-Box. The argument weighting may be 0, and the OBB is computed using a Cloud-Point approach, or 1, and it is computed using a Surface-Weighting approach. If weighting=1, then the provided array must be a surface composed of triangles.

Exists also as in place version (_BB) that modifies a and returns None.

Parameters

- **a** (array or pyTree) – input mesh
- **method** (string) – choice between axis-aligned or oriented bounding box
- **weighting** (integer) – activation key for surface weighting approach

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Bounding box generation (array):

```
# - BB (array) -
import Generator as G
import Converter as C
import Geom as D

s = D.circle((0,0,0), 1., N=100)
a = G.BB(s)
C.convertArrays2File([a,s], 'out.plt')
```

- Bounding box generation (pyTree):

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

s = D.circle((0,0,0), 1., N=100)
a = G.BB(s); a[0] = 'bbox'
C.convertPyTree2File([s,a], 'out.cgns')
```

Generator.**CEBBIntersection**(*a1*, *a2*, *tol=1.e-10*)

Test the Cartesian Elements Bounding Box (CEBB) intersection between a1 and a2. Tolerance is a float given by tol. Return 0 if no intersection, 1 otherwise.

Parameters

- **a1** (array or pyTree) – input mesh
- **a2** (array or pyTree) – input mesh
- **tol** (float) – tolerance of intersection

Returns 0 if no intersection, 1 otherwise

Return type integer

Example of use:

- Intersection by Cartesian elements bounding box between two meshes (array):

```
# - CEBBIntersection (array) -
import Generator as G
import Transform as T
ni = 11; nj = 3; nk = 11
a1 = G.cart((0.,0.,0.), (0.1,0.1,0.2),(ni, nj,nk))
a2 = G.cart((1.,0.,0.), (0.1,0.1,0.2),(ni, nj,nk))
a2 = T.rotate(a2, (0,0,0), (0,0,1), 12.)
print(G.CEBBIntersection(a1, a2))
```

- Intersection by Cartesian elements bounding box between two meshes (pyTree):

```
# - CEBBIntersection (pyTree)-
import Generator.PyTree as G
import Transform.PyTree as T

ni = 11; nj = 3; nk = 11
a1 = G.cart((0.,0.,0.), (0.1,0.1,0.2),(ni, nj,nk))
a2 = G.cart((1.,0.,0.), (0.1,0.1,0.2),(ni, nj,nk))
a2 = T.rotate(a2, (0,0,0), (0,0,1), 12.)
print(G.CEBBIntersection(a1, a2))
```

Generator.**bboxIntersection**(*a1*, *a2*, *tol=1.e-6*, *isBB=False*, *method='AABB'*)

Test if a1 and a2 intersects. Three options are available: method='AABB' (intersection between two Axis-Aligned Bounding Boxes, by default); method='OBB' (intersection between two Oriented Bounding Boxes, the most general case); method='AABBOBB' (intersection between an AABB -a1- and an OBB -a2-).; If a1 and a2 are directly the corresponding bounding boxes, the user may switch isBB=True in order to avoid recalculating them. Return 0 if no intersection, 1 otherwise.

Exists also as in place version (_bboxIntersection) that modifies a1 and returns None.

Parameters

- **a1** (array or pyTree) – input mesh
- **a2** (array or pyTree) – input mesh
- **tol** (float) – tolerance of intersection
- **isBB** (boolean) – activation key if meshes already are bounding boxes
- **method** (string) – intersection method

Returns 0 if no intersection, 1 otherwise

Return type integer

Example of use:

- Intersection by bounding box between two meshes (array):

```
# - bboxIntersection (array) -
import Generator as G
ni = 11; nj = 3; nk = 11
a1 = G.cart((0.,0.,0.), (0.1,0.1,0.2),(ni, nj,nk))
a2 = G.cart((0.5,0.05,0.01), (0.1,0.1,0.2),(ni, nj,nk))
intersect = G.bboxIntersection(a1,a2); print(intersect)
```

- Intersection by bounding box between two meshes (pyTree):

```
# - bboxIntersection (pyTree) -
import Generator.PyTree as G
ni = 11; nj = 3; nk = 11
a1 = G.cart((0.,0.,0.), (0.1,0.1,0.2),(ni, nj,nk))
a2 = G.cart((0.5,0.05,0.01), (0.1,0.1,0.2),(ni, nj,nk))
intersect = G.bboxIntersection(a1, a2); print(intersect)
```

Generator.**checkPointInCEBB**(*a*, *(x, y, z)*)

Test if a given point is in the CEBB of a.

Parameters

- **a** (array or pyTree) – input mesh
- **(x,y,z)** (3-tuple of floats) – coordinates of point

Returns 0 if point is not in the CEBB of a, 1 otherwise

Return type integer

Example of use:

- Detection of point location in the bounding box of a mesh (array):

```
# - checkPointInCEBB (array) -
import Generator as G
import Transform as T

Ni = 20; Nj = 20
a1 = G.cart((0,0,0),(1./Ni,0.5/Nj,1),(Ni,Nj,2))
a2 = G.cart((-0.1,0,0),(0.5/Ni, 0.5/Nj, 1), (Ni,Nj,2))
a2 = T.rotate(a2, (-0.1,0,0), (0,0,1), 0.22)

# Check if point is in CEBB of mesh2
val = G.checkPointInCEBB(a2, (0.04839, 0.03873, 0.5)); print(val)
```

- Detection of point location in the bounding box of a mesh (pyTree):

```
# - checkPointInCEBB (pyTree) -
import Generator.PyTree as G
import Transform.PyTree as T

Ni = 20; Nj = 20
a2 = G.cart((-0.1,0,0),(0.5/Ni, 0.5/Nj, 1), (Ni,Nj,2))
a2 = T.rotate(a2, (-0.1,0,0), (0,0,1), 0.22)
```

```
# Check if point is in CEBB of a2
val = G.checkPointInCEBB(a2, (0.04839, 0.03873, 0.5)); print(val)
```

Generator.**getVolumeMap**(*a*)

Return the volume field of an array. Volume is located at centers of cells.

Exists also as in place version (_getVolumeMap) that modifies a and returns None.

Parameters **a** (array or pyTree) – input volume or surface mesh

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Computation of cells volume (array):

```
# - getVolumeMap (array) -
import Generator as G
import Converter as C

a = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,10,3))
vol = G.getVolumeMap(a)
vol = C.center2Node(vol); vol = C.addVars([a,  vol])
C.convertArrays2File(vol, "out.plt")
```

- Computation of cells volume (pyTree):

```
# - getVolumeMap (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C

a = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,10,3))
a = G.getVolumeMap(a)
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**getNormalMap**(*a*)

Return the surface normals field of a surface array. It is located at centers of cells.

Exists also as in place version (_getNormalMap) that modifies a and returns None.

Parameters **a** (array or pyTree) – input surface mesh

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Computation of surface normals (array):

```
# - getNormalMap (array) -
import Geom as D
import Generator as G
import Converter as C

# 2D structured
a = D.sphere((0,0,0), 1, 50)
n = G.getNormalMap(a)
n = C.center2Node(n);n = C.addVars([a, n])
C.convertArrays2File([n], 'out1.plt')

# 2D unstructured
a = D.sphere((0,0,0), 1, 50)
a = C.convertArray2Tetra(a)
n = G.getNormalMap(a)
n = C.center2Node(n);n = C.addVars([a, n])
C.convertArrays2File([n], 'out2.plt')
```

- Computation of surface normals (pyTree):

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

a = D.sphere((0,0,0), 1, 50)
a = G.getNormalMap(a)
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**getSmoothNormalMap**(*a*, *niter=2*, *eps=0.4*)

Return the smoothed surface normals field of a surface array, located at nodes. niter is the number of smoothing operations, and eps is a smoothing weight.

Exists also as in place version (_getSmoothNormalMap) that modifies a and returns None.

Parameters

- **a** (array or pyTree) – input surface mesh
- **niter** (integer) – smoothing iterations number
- **eps** (float) – smoothing weight

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Computation of surface smoothed normals (array):

```
# - getSmoothNormalMap (array) -
import Converter as C
import Generator as G
import Transform as T

a = G.cart((0.,0.,0.),(1.,1.,1.),(10,10,1))
b = G.cart((0.,0.,0.),(1.,1.,1.),(1,10,10))
b = T.rotate(b,(0.,0.,0.),(0.,1.,0.),45.)
c = C.convertArray2Hexa([a,b])
c = T.join(c); c = T.reorder(c,(1,))
c = T.rotate(c,(0.,0.,0.),(0.,1.,0.),15.)
s = G.getSmoothNormalMap(c, niter=4)
c = C.addVars([c,s])
C.convertArrays2File(c, "out.plt")
```

- Computation of surface smoothednormals (pyTree):

```
# - getSmoothNormalMap (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Transform.PyTree as T

a = G.cart((0.,0.,0.),(1.,1.,1.),(10,10,1))
b = G.cart((0.,0.,0.),(1.,1.,1.),(1,10,10))
b = T.rotate(b,(0.,0.,0.),(0.,1.,0.),45.)
c = C.convertArray2Hexa([a,b])
c = T.join(c); c = T.reorder(c,(1,))
c = T.rotate(c,(0.,0.,0.),(0.,1.,0.),15.)
c = G.getSmoothNormalMap(c,niter=4)
C.convertPyTree2File(c, "out.cgns")
```

Generator.**getOrthogonalityMap**(*a*)

Return the orthogonality map of an array. The orthogonality map corresponds to the maximum deviation of all dihedral angles of an element. The orthogonality map is expressed in degree and located at centers.

Exists also as in place version (_getOrthogonalityMap) that modifies a and returns None.

Parameters **a** (array or pyTree) – input mesh

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Computation of cells orthogonality (array):

```
# - getOrthogonalityMap (array) -
import Generator as G
import Converter as C

a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 10., (50,50,10))
ac = C.node2Center(a)
ortho = G.getOrthogonalityMap(a)
ortho = C.addVars([ac,  ortho])
C.convertArrays2File([ortho], "out.plt")
```

- Computation of cells orthogonality (pyTree):

```
# - getOrthogonalityMap (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C

a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 10., (50,50,10))
a = G.getOrthogonalityMap(a)
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**getRegularityMap**(*a*)

Return the regularity map of an array. The regularity map corresponds to the maximum deviation of the volume ratio of an element and all its neigbouring cells. The regularity map is located at centers.

Exists also as in place version (_getRegularityMap) that modifies a and returns None.

Parameters **a** (array or pyTree) – input mesh

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Computation of cells regularity (array):

```
# - getRegularityMap (array) -
import Generator as G
```

```
import Converter as C

a = G.cart( (0,0,0), (1,1,1), (50,50,1))
a = G.enforceX(a, 25, 0.1, 10, 10)
ac = C.node2Center(a)
reg = G.getRegularityMap(a)
reg = C.addVars([ac,  reg])
C.convertArrays2File([reg], "out.plt")
```

- Computation of cells regularity (pyTree):

```
# - getRegularityMapPT (array) -
import Generator.PyTree as G
import Converter.PyTree as C

a = G.cart((0,0,0), (1,1,1), (50,50,1))
a = G.enforceX(a, 25, 0.1, 10, 10)
a = G.getRegularityMap(a)
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**getTriQualityMap**(*a*)

Return the quality map of a TRI array. The triangle quality is a value between 0. (degenerated triangle) and 1. (equilateral triangle). The quality map is located at centers.

Exists also as in place version (_getTriQualityMap) that modifies a and returns None.

Parameters **a** (array or pyTree) – input surface TRI mesh

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Computation of triangles quality (array):

```
# - getTriQualitylityMap (array) -
import Generator as G
import Converter as C
import Geom as D

a = D.sphere( (0,0,0), 1, N=10 )
a = C.convertArray2Tetra(a); a = G.close(a)
n = G.getTriQualityMap(a)
n = C.center2Node(n); n = C.addVars([a, n])
C.convertArrays2File([n], "out.plt")
```

- Computation of triangles quality (pyTree):

```
# - getTriQualitylityMap (PyTree) -
import Generator.PyTree as G
import Converter.PyTree as C
import Geom.PyTree as D

a = D.sphere((0,0,0), 1, N=10)
a = C.convertArray2Tetra(a)
a = G.close(a)
t = C.newPyTree(['Base',2,a])
t = G.getTriQualityMap(t)
C.convertPyTree2File(t, 'out.cgns')
```

Generator.**getCellPlanarity**(*a*)

Return a measure of cell planarity for each cell. It is located at centers of cells.

Exists also as in place version (_getCellPlanarity) that modifies a and returns None.

Parameters **a** (array or pyTree) – input surface mesh

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Computation of cells planarity (array):

```
# - getCellPlanarity (array) -
import Converter as C
import Generator as G
import Geom as D

a = D.sphere( (0,0,0), 1., 10)
p = G.getCellPlanarity(a)
p = C.center2Node(p); a = C.addVars([a, p])
C.convertArrays2File([a], 'out.plt')
```

- Computation of cells planarity (pyTree):

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

```
a = D.sphere((0,0,0), 1., 10)
a = G.getCellPlanarity(a)
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**getCircumCircleMap**(*a*)

Return the map of circum circle radius of any cell of a 'TRI' array.

Exists also as in place version (_getCircumCircleMap) that modifies a and returns None.

Parameters **a** (array or pyTree) – input surface mesh

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Computation of cells circumscribed circle radius (array):

```
# - getCircumCircleMap (array) -
import Geom as D
import Generator as G
import Converter as C

a = D.sphere((0,0,0), 1, 50)
a = C.convertArray2Tetra(a)
n = G.getCircumCircleMap(a)
n = C.center2Node(n); n = C.addVars([a, n])
C.convertArrays2File([n], "out.plt")
```

- Computation of cells circumscribed circle radius (pyTree):

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

a = D.sphere((0,0,0), 1, 50)
a = C.convertArray2Tetra(a)
t = C.newPyTree(['Base',2,a])
t = G.getCircumCircleMap(t)
C.convertPyTree2File(t, 'out.cgns')
```

Generator.**getInCircleMap**(*a*)

Return the map of inscribed circle radius of any cell of a 'TRI' array.

Exists also as in place version (_getInCircleMap) that modifies a and returns None.

Parameters **a** (array or pyTree) – input surface mesh

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Computation of cells inscribed circle radius (array):

```
# - getInCircleMap (array) -
import Geom as D
import Generator as G
import Converter as C

a = D.sphere((0,0,0), 1, 50)
a = C.convertArray2Tetra(a)
n = G.getInCircleMap(a)
n = C.center2Node(n); n = C.addVars([a, n])
C.convertArrays2File([n], "out.plt")
```

- Computation of cells inscribed circle radius (pyTree):

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

a = D.sphere((0,0,0), 1, 50)
a = C.convertArray2Tetra(a)
t = C.newPyTree(['Base',2,a])
t = G.getInCircleMap(t)
C.convertPyTree2File(t, 'out.cgns')
```

Generator.**getEdgeRatio**(*a*)

Return the ratio between the longest and the smallest edges of a cell.

Exists also as in place version (_getEdgeRatio) that modifies a and returns None.

Parameters **a** (array or pyTree) – input mesh

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Computation of maximum edge ratio of cells (array):

```
# - getEdgeRatio(array) -
import Generator as G
import Converter as C

a = G.cart((0.,0.,0.), (0.1,0.1,0.1), (11,11,11))
a = G.enforcePlusX(a,1e-6,(5,50))
r = G.getEdgeRatio(a)
r = C.center2Node(r); r = C.addVars([a,r])
C.convertArrays2File([r], "out.plt")
```

- Computation of maximum edge ratio of cells (pyTree):

```
# - getEdgeRatio(pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C

a = G.cart((0.,0.,0.), (0.1,0.1,0.1), (11,11,11))
a = G.enforcePlusX(a,1e-6,(5,50))
a = G.getEdgeRatio(a)
C.convertPyTree2File(a, "out.cgns")
```

Generator.**getMaxLength**(*a*)

Return the length of the longer edge of each cell.

Exists also as in place version (_getMaxLength) that modifies a and returns None.

Parameters **a** (array or pyTree) – input mesh

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Computation of maximum edge length of cells (array):

```
# - getMaxLength(array) -
import Generator as G
import Converter as C

a = G.cart((0.,0.,0.), (0.1,0.1,0.1), (11,11,11))
a = G.enforcePlusX(a,1e-6,(5,50))
r = G.getMaxLength(a)
```

```
r = C.center2Node(r); r = C.addVars([a,r])
C.convertArrays2File([r], "out.plt")
```

- Computation of maximum edge length of cells (pyTree):

```
# - getMaxLength(pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C

a = G.cart((0.,0.,0.), (0.1,0.1,0.1), (11,11,11))
a = G.enforcePlusX(a,1e-6,(5,50))
a = G.getMaxLength(a)
C.convertPyTree2File(a, "out.cgns")
```

3.7 Operations on distributions

Generator.**enforceX**(*a*, *x0*, *enforcedh*, *(supp*, *add)*)

Enforce a region around a line x=x0. The size of the cell around the line is enforcedh. "supp" points are suppressed from the starting distribution on the left and right side. "add" points are added on the left and add points are added on the right. Add exactely add points. Adjust add in order to have a monotonic distribution with: Generator.enforceX(a, x0, enforcedh, supp, add). Exists also for Y and Z directions: Generator.enforceY, Generator.enforceZ.

Parameters

- **a** (array or pyTree) – input structured mesh
- **x0** (float) – X-coordinate for refinement
- **enforcedh** (float) – cell size near refinement
- **supp** (integer) – number of suppressed points
- **add** (integer) – number of added points

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Structured mesh refinement around a line x=x0 (array):

```
# - enforceX (array) -
import Generator as G
import Converter as C
```

```
Ni = 50; Nj = 50
a = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,1))
# Monotonic distribution
b = G.enforceX(a, 0.3, 0.001, (13,25))
C.convertArrays2File([b], "out.plt")
```

- Structured mesh refinement around a line x=x0 (pyTree):

```
# - enforceX (pyTree) -
import Generator.PyTree as G
import Converter.PyTree as C

Ni = 50; Nj = 50; Nk = 2
a = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,Nk))
a = G.enforceX(a, 0.3, 0.001, (13,25))
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**enforceMoinsX**(*a*, *enforcedh*, *(supp*, *add)*)

Same as before but with a one sided distribution (left). This can be usefull to create a boundary layer distribution in an Euler mesh. Adjust add in order to have a monotonic distribution with: Generator.enforceMoinsX(a, enforcedh, supp, add). Exists also for Y and Z directions: Generator.enforceMoinsY, Generator.enforceMoinsZ.

Parameters

- **a** (`array or pyTree`) – input structured mesh
- **enforcedh** (`float`) – cell size near refinement
- **supp** (`integer`) – number of suppressed points
- **add** (`integer`) – number of added points

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Structured mesh refinement at left side (array):

```
# - enforceMoinsX (array) -
import Generator as G
import Converter as C

Ni = 50; Nj = 50
a = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,1))
```

```
b = G.enforceMoinsX(a, 1.e-3, (10,15))
C.convertArrays2File([b], "out.plt")
```

- Structured mesh refinement at left side (pyTree):

```
# - enforceMoinsX (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G

Ni = 50; Nj = 50; Nk = 1
a = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,Nk))
b = G.enforceMoinsX(a, 1.e-3, (10,15))
C.convertPyTree2File(b, 'out.cgns')
```

Generator.**enforcePlusX**(*a*, *enforcedh*, *(supp*, *add)*)

Same as before but with a one sided distribution (right). Adjust add in order to have a monotonic distribution with: Generator.enforcePlusX(a, x0, enforcedh, supp, add). Exists also for Y and Z directions: Generator.enforceMoinsY, Generator.enforcePlusZ.

Parameters

- **a** (array or pyTree) – input structured mesh
- **enforcedh** (float) – cell size near refinement
- **supp** (integer) – number of suppressed points
- **add** (integer) – number of added points

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Structured mesh refinement at right side (array):

```
# - enforcePlusX (array) -
import Generator as G
import Converter as C

# Distribution
Ni = 50; Nj = 50
a = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,1))
b = G.enforcePlusX(a, 1.e-3, (10,20))
C.convertArrays2File([b], "out.plt")
```

- Structured mesh refinement at right side (pyTree):

```
# - enforcePlusX (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G

# Distribution
Ni = 50; Nj = 50
a = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,1))
b = G.enforcePlusX(a, 1.e-3, (10,20))
C.convertPyTree2File(b, 'out.cgns')
```

Generator.**enforceLine**(*a*, *line*, *enforcedh*, *(supp*, *add)*)

Enforce a curvilinear line defined by the array line in a distribution defined by the array a.

Parameters

- **a** (array or pyTree) – input 2D distribution
- **line** (array) – line
- **enforcedh** (float) – cell size near refinement
- **supp** (integer) – number of suppressed points
- **add** (integer) – number of added points

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Distribution refinement around a line (array):

```
# - enforceLine (array) -
import Generator as G
import Converter as C
import Geom as D

Ni = 50; Nj = 50
a = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,1))
b = D.line((0.,0.2,0.), (1.,0.2,0.), 20)
c = G.enforceLine(a, b, 0.01, (5,3))
C.convertArrays2File([c], 'out.plt')
```

- Distribution refinement around a line (pyTree):

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

Ni = 50; Nj = 50
a = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,1))
b = D.line((0.,0.2,0.), (1.,0.2,0.), 20)
a = G.enforceLine(a, b, 0.01, (5,3))
C.convertPyTree2File(a, 'out.cgns')
```

Generator.**enforcePoint**(*a*, *x0*)

Enforce a point in the distribution. The index of enforced point is returned.

Parameters

- **a** (array or pyTree) – input 2D distribution
- **x0** (float) – I-location of the refinement point

Returns index of enforced point

Return type integer

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Distribution refinement at a point (array):

```
# - enforcePoint (array) -
import Converter as C
import Generator as G

# distribution
Ni = 20; Nj = 20
a = G.cart((0,0,0), (1./(Ni-1),5./(Nj-1),1), (Ni,Nj,1))
b = G.enforcePoint(a, 0.5)
C.convertArrays2File([b], "out.plt")
```

- Distribution refinement at a point (pyTree):

```
# - enforcePoint (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
```

```
Ni = 20; Nj = 20
a = G.cart((0,0,0), (1./(Ni-1),5./(Nj-1),1), (Ni,Nj,1))
b = G.enforcePoint(a, 0.5)
C.convertPyTree2File(b, 'out.cgns')
```

Generator.**enforceCurvature**(*a*, *curve*, *power=0.5*)

Enforce the curvature of an i-curve in a distribution defined by a. Power reflects the power of stretching.

Parameters

- **a** (array or pyTree) – input 2D distribution
- **curve** (array) – reference curve for curvature
- **power** (float) – stretching ratio

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Distribution refinement wrt. a curve curvature (array):

```
# - enforceCurvature (array) -
import Geom as D
import Generator as G
import Converter as C
import Transform as T

# Naca profile with lines
a = D.naca(12., 501)
l1 = D.getLength(a)
a2 = D.line((1.,0.,0.),(2.,0.,0.), 500)
l2 = D.getLength(a2)
b = T.join(a, a2)
c = D.line((2.,0.,0.),(1.,0.,0.), 500)
res = T.join(c, b)

# Distribution on the profile
l = l1+2*l2
Ni = 100; Nj = 100
p1 = l2/l; p2 = (l2+l1)/l
h = (p2-p1)/(Ni-1)
distrib = G.cart((p1,0,0), (h, 0.25/Nj,1), (Ni,Nj,1))
distrib = G.enforceCurvature(distrib, res, 0.6)
C.convertArrays2File([distrib], "out.plt")
```

- Distribution refinement wrt. a curve curvature (pyTree):

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

a = D.naca(12., 501)

# Distribution on the profile
Ni = 20; Nj = 20; Nk = 1; h = 1./(Ni-1)
b = G.cart((0,0,0), (h, 0.25/Nj,1), (Ni,Nj,Nk))
b = G.enforceCurvature(b, a, 0.6)
C.convertPyTree2File(b, 'out.cgns')
```

Generator.**addPointInDistribution**(*a*, *ind*)

Add a point in a distribution at index ind.

Parameters

- **a** (array or pyTree) – input distribution
- **ind** (integer) – I-index of inserted point

Returns modified reference copy of a

Return type array or pyTree

Example of use:

- Point insertion in a distribution (array):

```
# - addPointInDistribution (array) -
import Generator as G
import Converter as C

# Distribution
Ni = 50; Nj = 50
a = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,1))
b = G.addPointInDistribution( a, Ni )
C.convertArrays2File([b], 'out.plt')
```

- Point insertion in a distribution (pyTree):

```
# - addPointInDistribution (pyTree)-
import Generator.PyTree as G
import Converter.PyTree as C
```

```
# Distribution
Ni = 50; Nj = 50
a = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,2))
b = G.addPointInDistribution(a, Ni)
C.convertPyTree2File(b, 'out.cgns')
```

CHAPTER

FOUR

INDEX

- genindex
- modindex
- search