Transform Documentation

Release 3.1

/ELSA/MU-09023/V3.1

May 28, 2020

CONTENTS

CHAPTER

ONE

PREAMBLE

Transform module performs simple transformations of meshes. It works on arrays (as defined in Converter documentation) or on CGNS/Python trees (pyTrees), if they provide grid coordinates. In the pyTree version, flow solution nodes and boundary conditions and grid connectivity are preserved if possible. In particular, splitting a mesh does not maintain the grid connectivity.

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

To use the module with the Converter.array interface:

```
import Transform as T
```

To use the module with the CGNS/Python interface:

```
import Transform.PyTree as T
```

CHAPTER

TWO

LIST OF FUNCTIONS

– Basic operations

Transform.oneovern(a, N[, add])	Take one over N points from mesh.
Transform.reorder(a, order)	Reorder the numerotation of mesh.
Transform.reorderAll(arrays[, dir])	Orientate normals of all surface blocks consistently in one direction (1) or the opposite (-1).
Transform.makeCartesianXYZ(a)	Reorder a Cartesian mesh in order to get i,j,k aligned with X,Y,Z.
Transform.makeDirect(a)	Reorder a structured mesh to make it direct.
Transform.addkplane(a[, N])	Add N k-plane(s) to a mesh.
Transform.collapse(a)	Collapse the smallest edge of each element for TRI arrays.
Transform.patch(a1, a2[, position, nodes])	Patch mesh2 defined by a2 in mesh1 defined by a1 at position (i,j,k).

– Mesh positioning

Transform.rotate(a, center, arg1[, arg2, ...])	Rotate a grid.
Transform.translate(a, transvect)	Translate a grid.

– Mesh transformation

Transform.cart2Cyl(a[, center, axis])	Transform a mesh defined in Cartesian coordinates into cylindrical coordinates.
Transform.homothety(a, center, alpha)	Make for a mesh defined by an array an homothety of center Xc and of factor alpha.

Continued on next page

Table 3 – continued from previous page

Transform.contract(a, center, dir1, dir2, alpha)	Contract a mesh around a plane defined by (center, dir1, dir2) and of factor alpha.
Transform.scale(a[, factor])	Scale a mesh following factor (constant) or (f1,f2,f3) following dir.
Transform.symetrize(a, point, vector1, vector2)	Make a symetry of mesh from plane passing by point and of director vector: vector1 and vector2.
Transform.perturbate(a, radius[, dim])	Perturbate a mesh randomly of radius Usage: perturbate(a, radius, dim)
Transform.smooth(a[, eps, niter, type, ...])	Smooth a mesh with a Laplacian.
Transform.dual(array[, extraPoints])	Returns the dual mesh of a conformal mesh.
Transform.breakElements(a)	Break an array (in general NGON) in a set of arrays of BAR, TRI, ...

– Mesh splitting and merging

Transform.subzone(array, minIndex[, ...])	Take a subzone of mesh.
Transform.join(array[, array2, arrayc, ...])	Join two arrays in one or join a list of arrays in one.
Transform.merge(A[, Ac, sizeMax, dir, tol, ...])	Merge a list of matching structured grids.
Transform.mergeCart(A[, sizeMax, tol])	Merge a list of Cartesian zones using the method of weakest descent.
Transform.splitNParts(arrays, N[, ...])	Split blocks in N blocks.
Transform.splitSize(array[, N, multigrid, ...])	Split a block until it has less than N points.
Transform.splitCurvatureAngle(array, sensibility)	Split a line following curvature angle.
Transform.splitCurvatureRadius(a[, Rs])	Return the indices of the array where the curvature radius is low.
Transform.splitConnexity(a)	Split array into connex zones.
Transform.splitMultiplePts(A[, dim])	Split any zone of A if it is connected to several blocks at a given border.
Transform.splitSharpEdges(array[, alphaRef])	Split array into smooth zones (angles between elements are less than alphaRef).
Transform.splitTBranches(array[, tol])	Split a BAR into a set of BARS at vertices where T branches exist.

Continued on next page

Table 4 – continued from previous page

Transform.splitManifold(array)	Split an unstructured mesh (only TRI or BAR currently) into several manifold pieces.
Transform.splitBAR(array, N[, N2])	Split BAR at index N (start 0).
Transform.splitTRI(array, idxList)	Split a TRI into several TRIs delimited by the input poly line defined by the lists of indices idxList.

– Mesh deformation

Transform.deform(a[, vector])	Deform surface by moving surface of the vector dx, dy, dz.
Transform.deformNormals(array, alpha[, niter])	Deform a a surface of alpha times the surface normals.
Transform.deformPoint(a, xyz, dxdydz, depth, ...)	Deform mesh by moving point (x,y,z) of a vector (dx, dy, dz).
Transform.deformMesh(a, surfDelta[, beta, type])	Deform a mesh wrt surfDelta defining surface grids and deformation vector on it.

– Mesh projections

Transform.projectAllDirs(arrays, surfaces[, ...])	Project points defined in arrays to surfaces according to the direction provided by vect.
Transform.projectDir(surfaces, arrays, dir)	Project surfaces onto surface arrays following dir.
Transform.projectOrtho(surfaces, arrays)	Project a list of zones surfaces onto surface arrays following normals.
Transform.projectOrthoSmooth(surfaces, arrays)	Project a list of zones surfaces onto surface arrays following normals.
Transform.projectRay(surfaces, arrays, P)	Project surfaces onto surface arrays using rays starting from P.

CHAPTER

THREE

CONTENTS

3.1 Basic operations

Transform.**oneovern**(*a*, *(Ni*, *Nj*, *Nk)*)

Extract every Ni,Nj,Nk points in the three directions of a structured mesh a.

Exists also as an in-place version (_oneovern) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones, base, pyTree]) – input data
- **(Ni,Nj,Nk)** (3-tuple of integers) – period of extraction in the three directions

Returns a coarsened structured mesh

Return type identical to input

Example of use:

- Extract every two points in a structured mesh (array):

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

a = G.cart((0,0,0), (1,1,1), (10,10,1))
a2 = T.oneovern(a, (2,2,2))
C.convertArrays2File(a2, "out.plt")
```

- Extract every two points in a structured mesh (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (10,10,1))
a2 = T.oneovern(a, (2,2,1)); a2[0] = 'cart2'
C.convertPyTree2File([a,a2], "out.cgns")
```

Transform.**reorder**(*a*, *dest*)

For a structured grid, change the (i,j,k) ordering of a. If you set dest=(i2,j2,k2) for a (i,j,k) mesh, going along i2 direction of the resulting mesh will be equivalent to go along i direction of initial mesh.

The transformation can be equivalently described by a matrix M, filled with a single non-zero value per line and column (equal to -1 or 1). Then, dest=(desti,destj,destk) means: M[abs(desti),1]=sign(desti); M[abs(destj),2]=sign(destj); M[abs(destk),3]=sign(destk).

For an unstructured 2D grid (TRI, QUAD, 2D NGON), order the element nodes such that all normals are oriented towards the same direction. If dest is set to (1,), all elements are oriented as element 0. If dest is (-1,), all elements are oriented in the opposite sense of element 0.

Exists also as an in-place version (_reorder) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones, base, pyTree]) – initial mesh
- **dest** (3-tuple of signed integers or a tuple of a single 1 or -1) – integers specifying transformation

Returns a reoriented mesh

Return type identical to input

Example of use:

- Reorder a mesh (array):

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

# Structured
```

(continues on next page)

(continued from previous page)

```
a = G.cart((0,0,0),(1,1,1),(5,7,9))
a = T.reorder(a, (3,-2,-1))
C.convertArrays2File(a, 'out1.plt')

# Unstructured
a = G.cartTetra((0,0,0),(1,1,1),(3,3,1))
a = T.reorder(a, (1,))
C.convertArrays2File(a, 'out2.plt')
```

- Reorder a mesh (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (8,9,20))
a = T.reorder(a, (2,1,-3))
C.convertPyTree2File(a, "out.cgns")
```

Transform.**reorderAll**(*a*, *dir=1*)

Order a set of surface grids a such that their normals points in the same direction. All the grids in a must be of same nature (structured or unstructured). Orientation of the first grid in the list is used to reorder the other grids. If dir=-1, the orientation is the opposite direction of the normals of the first grid.

In case of unstructured grids, reorientation is guaranteed to be outward (dir=1) if they represent a closed volume.

Exists also as an in-place version (_reorderAll) which modifies a and returns None.

Parameters

- **a** ([list of arrays] or [list of zones]) – initial set of surface grids
- **dir** (signed integer) – 1 (default), -1 for a reversed orientation

Returns a reoriented mesh

Return type identical to input

Example of use:

- Reorder a set of grids (array):

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

ni = 30; nj = 40
m1 = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,1))
m2 = T.rotate(m1, (0.2,0.2,0.), (0.,0.,1.), 15.)
m2 = T.reorder(m2,(-1,2,3))
a = [m1,m2]
a = T.reorderAll(a,1)
C.convertArrays2File(a, "out.plt")
```

- Reorder a set of grids (pyTree):

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

ni = 30; nj = 40; nk = 1
m1 = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk)); m1[0]='cart1'
m2 = T.rotate(m1, (0.2,0.2,0.), (0.,0.,1.), 15.)
m2 = T.reorder(m2,(-1,2,3)); m2[0]='cart2'
t = C.newPyTree(['Base',2,[m1,m2]])
t = T.reorderAll(t, 1)
C.convertPyTree2File(t, "out.cgns")
```

Transform.**makeCartesianXYZ**(*a*)

Reorder a structured Cartesian mesh in order to get i,j,k aligned with X,Y,Z respectively. Exists also as an in-place version (_makeCartesianXYZ) which modifies a and returns None.

Parameters **a** ([array, list of arrays] or [zone, list of zones, base, pyTree]) – Cartesian mesh with (i,j,k) not ordered following (X,Y,Z)

Returns a Cartesian mesh such that direction i is aligned with (0X), ...

Return type identical to input

Example of use:

- Align a Cartesian mesh with XYZ (array):

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

a = G.cart((0.,0.,0.),(1.,1.,1.),(11,12,13))
a = T.reorder(a, (3,2,-1))
a = T.makeCartesianXYZ(a)
C.convertArrays2File(a, 'out.plt')
```

- Align a Cartesian mesh with XYZ (pyTree):

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

a = G.cart((0.,0.,0.),(1.,1.,1.),(11,12,13))
a = T.reorder(a, (3,2,-1))
a = T.makeCartesianXYZ(a)
C.convertPyTree2File(a, 'out.cgns')
```

Transform.**makeDirect**(*a*)

Reorder an indirect structured mesh to get a direct mesh. Exists also as an in-place version (_makeDirect) which modifies a and returns None.

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

Returns a direct structured mesh

Return type identical to input

Example of use:

- Make a mesh direct (array):

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

a = G.cart((0.,0.,0.),(1.,1.,1.),(10,10,10))
a = T.reorder(a, (1,2,-3)) # indirect now
a = T.makeDirect(a)
C.convertArrays2File(a, 'out.plt')
```

- Make a mesh direct (pyTree):

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

a = G.cart((0.,0.,0.),(1.,1.,1.),(10,10,10))
a = T.reorder(a, (1,2,-3)) # indirect now
a = T.makeDirect(a)
C.convertPyTree2File(a, 'out.cgns')
```

Transform.**addkplane**(*a*, *N=1*)

Add one or more planes at constant heights in z: z0+1, ...,z0+N. Exists also as an in-place version (_addkplane) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones, base, pyTree]) – any mesh
- **N** (integer) – number of layers in the k direction to be added

Returns expanded mesh

Return type identical to input

Example of use:

- Add a k-plane in direction (Oz) (array):

```
# - addkplane (array) -
import Geom as D
import Transform as T
import Converter as C

a = D.naca(12., 501)
a = T.addkplane(a)
C.convertArrays2File(a, "out.plt")
```

- Add a k-plane in direction (Oz) (pyTree):

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

(continues on next page)

(continued from previous page)

```
a = G.cart((0.,0.,0.),(0.1,0.1,1.),(10,10,2))
a = T.addkplane(a)
C.convertPyTree2File(a, 'out.cgns')
```

Transform.**collapse**(*a*)

Collapse the smallest edges of each element of a triangular mesh. Return each element as a BAR. Exists also as an in-place version (_collapse) which modifies a and returns None.

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

Returns a BAR mesh

Return type identical to input

Example of use:

- Collapse smallest edge of a triangular mesh (array):

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

a = G.cartTetra((0.,0.,0.),(0.1,0.01,1.),(20,2,1))
b = T.collapse(a)
C.convertArrays2File([a,b], "out.plt")
```

- Collapse smallest edge of a triangular mesh (pyTree):

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

a = G.cartTetra((0.,0.,0.),(0.1,0.01,1.),(20,2,1))
b = T.collapse(a)
C.convertPyTree2File(b, "out.cgns")
```

Transform.**patch**(*a*, *b*, *position=None*, *nodes=None*)

For a structured mesh a, patch (replace) a structured mesh b from starting point position=(i,j,k) of a. Coordinates and fields are replaced.

For an unstructured mesh a, patch an unstructured mesh (of same type) by replacing the nodes of indices nodes.

Exists also as an in-place version (_patch) which modifies a and returns None.

Parameters

- **a** (array or zone) – initial mesh
- **b** (array or zone) – patch mesh
- **position** (3-tuple of integers) – indices starting from 1 of the starting node to be replaced in a
- **nodes** (numpy array of integers (starting from 1)) – list of nodes of the unstructured mesh a to be replaced

Returns a modified zone

Return type an array or a zone

Example of use:

- Patch a mesh in another one (array):

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

c1 = G.cart((0,0,0), (0.01,0.01,1), (201,101,1))
c2 = G.cart((0,0,0), (0.01,0.01,1), (51,81,1))
c2 = T.rotate(c2, (0,0,0),(0,0,1),0.2)
c3 = G.cart((0.0,1.,0), (0.01,0.01,1), (101,1,1))
c3 = T.rotate(c3, (0,0,0),(0,0,1),0.3)
# patch a region at given position
a = T.patch(c1, c2, position=(1,1,1))
# patch some nodes
nodes = numpy.arange(20100, 20201, dtype=numpy.int32)
b = T.patch(c1, c3, nodes=nodes)
C.convertArrays2File([a,b], 'out.plt')
```

- Patch a mesh in another one (pyTree):

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

(continues on next page)

(continued from previous page)

```
c1 = G.cart((0,0,0), (0.01,0.01,1), (201,101,1))
c2 = G.cart((0,0,0), (0.01,0.01,1), (51,81,1))
c2 = T.rotate(c2, (0,0,0),(0,0,1),0.2)
a = T.patch(c1, c2, (1,1,1))
C.convertPyTree2File(a, 'out.cgns')
```

3.2 Mesh positioning

Transform.**rotate**(*a, C, arg1, arg2=None, vectors=[['VelocityX','VelocityY','VelocityZ'],['MomentumX','MomentumY','MomentumZ']]*)

Rotate a mesh. Rotation can be also applied on some vector fields (e.g. velocity and momentum). If the vector field is located at cell centers, then each vector component name must be prefixed by 'centers:'.

Exists also as an in-place version (_rotate) which modifies a and returns None.

Rotation parameters can be specified either by:

- a rotation axis (arg1) and a rotation angle in degrees (arg2)
- two axes (arg1 and arg2): axis arg1 is rotated into axis arg2
- three Euler angles in degrees arg1=(alpha, beta, gamma). alpha is a rotation along X (Ox->Ox, Oy->Oy1, Oz->Oz1), beta is a rotation along Y (Ox1->Ox2, Oy1->Oy1, Oz1->Oz2), gamma is a rotation along Z (Ox2->Ox3, Oy2->Oy3, Oz2->Oz2):

Parameters

- **a** (`[array, list of arrays] or [zone, list of zones, base, pyTree]`) – mesh
- **C** (`3-tuple of floats`) – center of rotation
- **arg1** (`3-tuple of floats or 3-tuple of 3-tuple of floats`) – rotation axis or original axis or rotation angles (in degrees)
- **arg2** (`float or 3-tuple of floats or None`) – angle of rotation (in degrees) or destination axis or None
- **vectors** (`[list of list of strings]`) – for each vector, list of the names of the vector components

Returns mesh after rotation

Return type identical to input

Example of use:

- Rotate a mesh (array):

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

a = G.cart((0,0,0), (1,1,1), (10,10,1))
# Rotate with an axis and an angle
b = T.rotate(a, (0.,0.,0.), (0.,0.,1.), 30.)
# Rotate with axis transformations
c = T.rotate(a, (0.,0.,0.), ((1.,0.,0.),(0,1,0),(0,0,1)),
             ((1,1,0), (1,-1,0), (0,0,1)) )
# Rotate with three angles
d = T.rotate(a, (0.,0.,0.), (90.,0.,0.))
C.convertArrays2File([a,d], 'out.plt')
```

- Rotate a mesh (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (10,10,2))
# Rotate with an axis and an angle
b = T.rotate(a, (0.,0.,0.), (0.,0.,1.), 30.); b[0] = 'cartRot1'
# Rotate with two axis
c = T.rotate(a, (0.,0.,0.), ((1.,0.,0.),(0,1,0),(0,0,1)),
             ((1,1,0), (1,-1,0), (0,0,1)) ); c[0] = 'cartRot2'
# Rotate with three angles
c = T.rotate(a, (0.,0.,0.), (0,0,90)); c[0] = 'cartRot3'
C.convertPyTree2File([a,b,c], 'out.cgns')
```

Transform.**translate**(*a*, *T*)

Translate a mesh of vector T=(tx,ty,tz).

Exists also as an in-place version (_translate) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones, base, pyTree]) – mesh
- **T** (3-tuple of floats) – translation vector

Returns mesh after translation

Return type identical to input

Example of use:

- Translate a mesh (array):

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

a = G.cart((0,0,0), (1,1,1), (10,10,1))
b = T.translate(a, (-1.,0.,0.))
C.convertArrays2File([a,b], 'out.plt')
```

- Translate a mesh (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (10,10,3))
T._translate(a, (10.,0.,0.))
C.convertPyTree2File(a, 'out.cgns')
```

3.3 Mesh transformation

`Transform.`**`cart2Cyl`**(*a*, *C*, *AXIS*)

Convert a mesh in Cartesian coordinates into a mesh in cylindrical coordinates. One of the Cartesian axes, defined by parameter AXIS, must be the revolution axis of the cylindrical frame. AXIS can be one of (0,0,1), (1,0,0) or (0,1,0).

Exists also as an in-place version (_cart2Cyl) which modifies a and returns None.

Parameters

- **a** (`[array, list of arrays] or [zone, list of zone, base, pyTree]`) – mesh with coordinates defined in the Cartesian frame
- **C** (`3-tuple of floats`) – center of revolution
- **AXIS** (`3-tuple of floats`) – revolution axis

Returns mesh with coordinates in the cylindrical frame

Return type identical to input

Example of use:

- Cart2Cyl a mesh (array):

```
# - cart2Cyl (array) -
import Transform as T
import Generator as G
import Converter as C
a = G.cylinder((0.,0.,0.), 0.5, 1., 0., 360, 1., (360,20,10))
a = T.cart2Cyl(a, (0.,0.,0.),(0,0,1))
C.convertArrays2File(a, 'out.plt')
```

- Cart2Cyl a mesh (pyTree):

```
# - cart2Cyl (pyTree) -
import Transform.PyTree as T
import Generator.PyTree as G
import Converter.PyTree as C
a = G.cylinder((0.,0.,0.), 0.5, 1., 0., 360., 1., (360,20,10))
T._cart2Cyl(a, (0.,0.,0.),(0,0,1))
C.convertPyTree2File(a, 'out.cgns')
```

Transform.**homothety**(*a*, *C*, *alpha*)

Apply an homothety of center C and a factor alpha to a mesh a.

Exists also as an in-place version (_homothety) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones, base, pyTree]) – mesh
- **C** (3-tuple of floats) – center of homothety
- **alpha** (float) – homothety factor

Returns mesh after homothety

Return type identical to input

Example of use:

- Homothety a mesh (array):

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

(continues on next page)

(continued from previous page)

```
import Converter as C

a = G.cart((0,0,0), (1,1,1), (10,10,1))
b = T.homothety(a, (0.,0.,0.), 2.)
C.convertArrays2File([a,b], 'out.plt')
```

- Homothety a mesh (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (10,10,10))
b = T.homothety(a, (0.,0.,0.), 2.); b[0] = 'cart2'
C.convertPyTree2File([a,b], "out.cgns")
```

Transform.**contract**(*a*, *C*, *dir1*, *dir2*, *alpha*)

Make a contraction of factor alpha of a mesh with respect to a plane defined by a point C and vectors dir1 and dir2.

Exists also as an in-place version (_contract) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones, base, pyTree]) – mesh
- **C** (3-tuple of floats) – point of the contraction plane
- **dir1** (3-tuple of floats) – first vector defining the plane
- **dir2** (3-tuple of floats) – second vector defining the plane
- **alpha** (float) – contraction factor

Returns mesh after contraction

Return type identical to input

Example of use:

- Contract a mesh (array):

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

(continues on next page)

(continued from previous page)

```
import Converter as C

a = G.cart((0,0,0), (1,1,1), (10,10,10))
b = T.contract(a, (0.,0.,0.), (1,0,0), (0,1,0), 0.1)
C.convertArrays2File([a,b], 'out.plt')
```

- Contract a mesh (pyTree):

```
# - contract (pytree) -
import Generator.PyTree as G
import Transform.PyTree as T
import Converter.PyTree as C

a = G.cart((0,0,0), (1,1,1), (10,10,10))
b = T.contract(a, (0.,0.,0.), (1,0,0), (0,1,0), 0.1); b[0]='cart2'
C.convertPyTree2File([a,b], 'out.cgns')
```

Transform.**scale**(*a*, *factor=1.*)

Scale a mesh of factor factor. If factor is a list of floats, scale with given factor for each canonical axis. The invariant point is the barycenter of a.

Exists also as an in-place version (_scale) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones, base, pyTree]) – mesh
- **factor** (float or list of 3 floats) – scaling factor

Returns mesh after scaling

Return type identical to input

Example of use:

- Scale a mesh (array):

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

a = G.cart((0,0,0), (1,1,1), (10,10,10))

# scale in all directions
```

(continues on next page)

(continued from previous page)

```
a = T.scale(a, factor=0.1)

# scale with different factors following directions
a = T.scale(a, factor=(0.1,0.2,0.3))

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

- Scale a mesh (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (10,10,10))

# scale in all directions
T._scale(a, factor=0.1)

# scale with different factors following directions
T._scale(a, factor=(0.1,0.2,0.3))

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

Transform.**symetrize**(*a*, *P*, *vector1*, *vector2*)

Symmetrize a mesh with respect to a plane defined by point P and vectors vector1 and vector2.

Exists also as an in-place version (_symetrize) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones, base, pyTree]) – mesh
- **C** (3-tuple of floats) – point of the symmetry plane
- **vector1** (3-tuple of floats) – first vector of the symetry plane
- **vector2** (3-tuple of floats) – second vector of the symetry plane

Returns mesh after symmetrization

Return type identical to input

Example of use:

- Symmetrize a mesh (array):

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

a = G.cart((0,0,0), (1,1,1), (10,10,1))
# Symetrize regarding plane (x,z)
b = T.symetrize(a, (0.,0.,0.), (1,0,0), (0,0,1))
C.convertArrays2File([a,b], "out.plt")
```

- Symmetrize a mesh (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (10,10,2))
# Symetrize regarding plane (x,z)
b = T.symetrize(a, (0.,0.,0.), (1,0,0), (0,0,1)); b[0]='cart2'
C.convertPyTree2File([a,b], "out.cgns")
```

Transform.**perturbate**(*a*, *radius*, *dim=3*)

Perturbate randomly a mesh a with given radius. Mesh points are modified aleatoirely in all directions, with a distance less or equal to radius. If dim=2, Z coordinates are fixed. If dim=1, only the X coordinates are modified.

Exists also as an in-place version (_perturbate) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones, base, pyTree]) – mesh
- **radius** (float) – radius of perturbation
- **dim** (integer) – to select if 1, 2 or the 3 coordinates are modified.

Returns mesh after perturbation

Return type identical to input

Example of use:

- Perturbate a mesh (array):

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

a = G.cart((0,0,0), (1,1,1), (10,10,1))
a = T.perturbate(a, 0.1)
C.convertArrays2File(a, "out.plt")
```

- Perturbate a mesh (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (10,10,2))
b = T.perturbate(a, 0.1); b[0]='cart2'
C.convertPyTree2File([a,b], "out.cgns")
```

Transform.**smooth**(*a*, *eps=0.5*, *niter=4*, *type=0*, *fixedConstraints=[]*, *projConstraints=[]*, *delta=1.*, *point=(0, 0, 0)*, *radius=-1.*)

Perform a Laplacian smoothing on a set of structured grids or an unstructured mesh ('QUAD', 'TRI') with a weight eps, and niter smoothing iterations. Type=0 means isotropic Laplacian, type=1 means scaled Laplacian, type=2 means taubin smoothing. Constraints can be defined in order to avoid smoothing of some points (for instance the exterior faces of a):

Exists also as an in-place version (_smooth) which modifies a and returns None.

Parameters

- **a** (array or zone) – input mesh
- **eps** (float) – smoother power
- **niter** (integer) – number of smoothing iterations
- **type** (integer) – type of smoothing algorithm
- **fixedConstraints** ([list of arrays] or [list of zones]) – set of fixed regions
- **projConstraints** ([list of arrays] or [list of zones]) – smoothed mesh projected on them
- **delta** (float) – strength of constraints

- **point** (3-tuple of float) – center of the region to be smoothed in case of local smoothing
- **radius** (float) – if local smoothing, radius of the region to be smoothed

Returns mesh after smoothing

Return type array or zone

Example of use:

- Smooth a mesh (array):

```
# - smooth (array) -
import Transform as T
import Converter as C
import Geom as D
a = D.sphere6((0,0,0), 1, N=20)
b = T.smooth(a, eps=0.5, niter=20)
C.convertArrays2File(a+b, "out.plt")
```

- Smooth a mesh (pyTree):

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

a = D.sphere6((0,0,0), 1, N=20)
b = T.smooth(a, eps=0.5, niter=20)
C.convertPyTree2File(b, "out.cgns")
```

Transform.**dual**(*a*, *extraPoints=1*)

Return the dual of a mesh a. If extraPoints=1, external face centers are added.

Exists also as an in-place version (_dual) which modifies a and returns None.

Parameters

- **a** (array or zone) – mesh
- **extraPoints** (integer) – 0/1 external face centers are added

Returns dual mesh

Return type array or zone

Example of use:

- Dual a mesh (array):

```
# - dual (arrays) -
import Converter as C
import Generator as G
import Transform as T

ni = 5; nj = 5; nk = 1
a = G.cart((0,0,0),(1,1,1),(ni,nj,nk))
a = C.convertArray2NGon(a); a = G.close(a)
res = T.dual(a)
C.convertArrays2File([res],'out.tp')
```

- Dual a mesh (pyTree):

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

ni = 5; nj = 5
a = G.cart((0,0,0),(1,1,1),(ni,nj,1))
a = C.convertArray2NGon(a); a = G.close(a)
res = T.dual(a)
C.convertPyTree2File(res, 'out.cgns')
```

Transform.**breakElements**(*a*)

Break a NGON mesh into a set of grids, each of them being a basic element grid (with a single connectivity).

Parameters **a** (array or zone) – NGON mesh

Returns list of grids of basic elements

Return type [list of arrays or list of zones]

Example of use:

- Break a NGON mesh into basic elements (array):

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

(continues on next page)

(continued from previous page)

```
a = G.cartTetra((0,0,0),(1,1,1),(3,3,2))
a = C.convertArray2NGon(a)
a = G.close(a)
b = G.cartNGon((2,0,0),(1,1,1),(3,2,2))
res = T.join(a,b)
res = T.breakElements(res)
C.convertArrays2File(res, 'out.plt')
```

- Break a NGON mesh into basic elements (pyTree):

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

a = G.cartTetra((0,0,0),(1,1,1),(3,3,2))
a = C.convertArray2NGon(a)
a = G.close(a)
b = G.cartNGon((2,0,0),(1,1,1),(3,3,1))
res = T.join(a,b)
res = T.breakElements(res)
C.convertPyTree2File(res, 'out.cgns')
```

3.4 Mesh splitting and merging

Transform.**subzone**(*a*, *minIndex*, *maxIndex=None*, *type=None*)

Extract a subzone.

Extract a subzone of a structured mesh a, where min and max ranges must be specified. Negative indices can be used (as in Python): -1 means max index:

```
b = T.subzone(a, (imin,jmin,kmin), (imax,jmax,kmax))
```

Extract a subzone of an unstructured mesh a, where the vertex list of the subzone must be specified (indices start at 1):

```
b = T.subzone(a, [1,2,...])
```

Extract a subzone of an unstructured mesh providing the indices of elements (index starts at 0):

```
b = T.subzone(a, [0,1,...], type='elements')
```

Extract a subzone of an unstructured array providing the indices of faces (for unstructured zones with basic elements: indFace=indElt*numberOfFaces+noFace, for NGON zones: use the natural face indexing, starting from 1):

```
b = T.subzone(a, [1,2,...], type='faces')
```

Parameters

- **a** (array or zone) – input data
- **minIndex** (3-tuple of integers) – (imin,jmin,kmin) for a structured grid, list of indices otherwise
- **maxIndex** (3-tuple of integers) – (imax,jmax,kmax) for a structured grid, None otherwise
- **type** (None or string) – type of subzone to perform (None, 'elements', 'faces')

Returns subzoned mesh

Return type identical to a

Example of use:

- Subzone (array):

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

# Structure
a = G.cart((0,0,0), (1,1,1), (10,20,10))
a = T.subzone(a, (3,3,3), (7,8,5))

# Structure avec indices negatif
e = G.cart((0,0,0), (1,1,1), (10,20,10))
e = T.subzone(e, (1,1,1), (-1,-1,-2)) # imax,jmax,kmax-1

# Non structure. Indices de noeuds -> retourne elts
b = G.cartTetra((0,0,0), (1,1,1), (2,2,2))
b = T.subzone(b, [2,6,8,5])

# Non structure. Indices d'elements -> Retourne elts
# Les indices d'elements commencent a 0
```

(continues on next page)

(continued from previous page)

```
c = G.cartTetra((0,0,0), (1,1,1), (5,5,5))
c = T.subzone(c, [0,1], type='elements')

# Non structure. Indices de faces:
# Pour les maillages BAR, TRI, TETRA... indFace=indElt*nbreFaces+noFace
# les noFace commence a 1
# Pour les NGONS... indices des faces
# -> Retourne les faces
d = G.cartTetra((0,0,0), (1,1,1), (2,2,2))
d = T.subzone(d, [1,2,3], type='faces')

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

- Subzone (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (10,20,1))
a = T.subzone(a, (3,3,1), (7,8,1))
C.convertPyTree2File(a, 'out.cgns')
```

`Transform.`**`join`**(*a*, *b=None*)

Join two zones in one (if possible) or join a list of zones in one zone (if possible). For the pyTree version, boundary conditions are maintained for structured grids only.

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

Returns unique joined zone

Return type array or zone

Example of use:

- Join (array):

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

a1 = G.cartTetra((0.,0.,0.), (1.,1.,1), (11,11,1))
```

(continues on next page)

(continued from previous page)

```
a2 = G.cartTetra((10.,0.,0.), (1.,1.,1), (10,10,1))
a = T.join(a1, a2)
C.convertArrays2File([a], 'out.plt')
```

- Join (pyTree):

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

a1 = D.naca(12., 5001)
a2 = D.line((1.,0.,0.),(20.,0.,0.),5001)
a = T.join(a1, a2)
C.convertPyTree2File(a, "out.cgns")
```

Transform.**merge**(*a*, *sizeMax=1000000000*, *dir=0*, *tol=1.e-10*, *alphaRef=180.*)

Join a set of zones such that a minimum number of zones is obtained at the end. Parameter sizeMax defines the maximum size of merged grids. dir is the constraint direction along which the merging is prefered. Default value is 0 (no prefered direction), 1 for i, 2 for j, 3 for k. alphaRef can be used for surface grids and avoids merging adjacent zones sharing an angle deviating of alphaRef to 180.

For the pyTree version, boundary conditions are maintained for structured grids only.

Parameters

- **a** (`[list of arrays] or [list of zones, base, pyTree]`) – list of grids
- **sizeMax** (`integer`) – maximum size of merged grids
- **dir** (`integer`) – direction of merging (structured grids only): 0:ijk; 1:i; 2:j; 3:k
- **tol** (`float`) – tolerance for abutting grids
- **alphaRef** (`float`) – angle max of deviation for abutting grids, above which grids are not merged (for surface grids only)

Returns list of merged grids

Return type [list of arrays] or [list of zones]

Example of use:

- Merge (array):

```
# - merge (array) -
import Converter as C
import Transform as T
import Geom as D
def f(t,u):
    x = t+u; y = t*t+1+u*u; z = u
    return (x,y,z)

a = D.surface(f)
b = T.splitSize(a, 100)
b = T.merge(b)
C.convertArrays2File(b, "out.plt")
```

- Merge (pyTree):

```
# - merge (pyTree) -
import Converter.PyTree as C
import Transform.PyTree as T
import Connector.PyTree as X
import Geom.PyTree as D

def f(t,u):
    x = t+u
    y = t*t+1+u*u
    z = u
    return (x,y,z)

a = D.surface(f)
b = T.splitSize(a, 100)
b = X.connectMatch(b, dim=2)
t = C.newPyTree(['Surface']); t[2][1][2] += b
b = T.merge(t)
t[2][1][2] = b
C.convertPyTree2File(t, "out.cgns")
```

Transform.**mergeCart**(*a*, *sizeMax=1000000000*, *tol=1.e-10*)

Merge a set of Cartesian grids. This function is similar to the function Transform.merge but is optimized for Cartesian grids.

Parameters

- **a** ([list of arrays] or [list of zones, base, pyTree]) – list of Cartesian grids
- **sizeMax** (integer) – maximum size of merged grids

- **tol** (float) – tolerance for abutting grids

Returns list of merged Cartesian grids

Return type [list of arrays] or [list of zones]

Example of use:

- Merge Cartesian grids (array):

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

dh = 0.1; n = 11
A = []
a1 = G.cart((0.,0.,0.),(dh,dh,dh),(n,n,n)); A.append(a1)
a2 = G.cart((1.,0.,0.),(dh,dh,dh),(n,n,n)); A.append(a2)
a3 = G.cart((1.,1.,0.),(dh,dh,dh),(n,n,n)); A.append(a3)
a4 = G.cart((0.,1.,0.),(dh,dh,dh),(n,n,n)); A.append(a4)
A[0] = T.oneovern(A[0],(2,2,2))
A[1] = T.oneovern(A[1],(2,2,2))
res = T.mergeCart(A)
C.convertArrays2File(res, "out.plt")
```

- Merge Cartesian grids (pyTree):

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

dh = 0.1; n = 11
A = []
a1 = G.cart((0.,0.,0.),(dh,dh,dh),(n,n,n)); a1 = T.oneovern(a1,(2,2,2))
a2 = G.cart((1.,0.,0.),(dh,dh,dh),(n,n,n)); a2 = T.oneovern(a2,(2,2,2))
a3 = G.cart((1.,1.,0.),(dh,dh,dh),(n,n,n))
a4 = G.cart((0.,1.,0.),(dh,dh,dh),(n,n,n))
A = [a1,a2,a3,a4]
for i in range(1,5): A[i-1][0] = 'cart'+str(i)

t = C.newPyTree(['Base']); t[2][1][2] += A
t[2][1][2] = T.mergeCart(t[2][1][2])
C.convertPyTree2File(t, "out.cgns")
```

Transform.**splitNParts**(*a, N, multigrid=0, dirs=[1,2,3],recoverBC=True*)

Split a set of M grids into N parts of same size roughly, provided M < N.

Argument multigrid enables to ensure the multigrid level by the splitting, provided the input grids are of that multigrid level.

For the pyTree version, boundary conditions and matching connectivity are split.

Exists also as in place version (_splitNParts) that modifies a and returns None. In this case, a must be a pyTree.

Parameters

- **a** (`[list of arrays] or [list of zones, base, pyTree]`) – list of grids
- **N** (`integer`) – number of grids after splitting
- **multigrid** (`integer`) – for structured grids only. 0: no constraints; 1: grids are 2n+1 per direction ; 2: grids are 4n+1 per direction
- **dirs** (`list of integers (possible values:1,2,3 or a combination of them)`) – directions where splitting is allowed (for structured grids only)
- **recoverBC** (`Boolean (True or False)`) – BCs are recovered after split (True) or not (False)

Returns list of splitted grids

Return type [list of arrays] or [list of zones]

Example of use:

- Split a mesh in N parts (array):

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

a = G.cart((0,0,0), (1,1,1), (101,101,41))
b = G.cart((10,0,0), (1,1,1), (121,61,81))
c = G.cart((20,0,0), (1,1,1), (101,61,131))
res = T.splitNParts([a,b,c], 32, multigrid=0, dirs=[1,2,3])

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

- Split a mesh in N parts (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (81,81,81))
b = G.cart((80,0,0), (1,1,1), (41,81,41))
t = C.newPyTree(['Base',a,b])
t = T.splitNParts(t, 10, multigrid=0, dirs=[1,2,3])
C.convertPyTree2File(t, 'out.cgns')
```

Transform.**splitSize**(*a, N, multigrid=0, dirs=[1,2,3], type=0, R=None, minPtsPerDir=5*)

- Split structured blocks if their number of points is greater than N.
- splitSize can also be used to split blocks in order to fit as better as possible on a number of R processors.

Argument multigrid enables to ensure the multigrid level by the splitting, provided the input grids are of that multigrid level.

For the pyTree version, boundary conditions and matching connectivity are split.

Exists also as in place version (_splitSize) that modifies a and returns None. In this case, a must be a pyTree.

Parameters

- **a** (`[list of arrays] or [list of zones, base, pyTree]`) – list of grids
- **N** (`integer`) – number of grids after splitting
- **multigrid** (`integer`) – for structured grids only. 0: no constraints; 1: grids are 2n+1 per direction ; 2: grids are 4n+1 per direction
- **dirs** (`list of integers (possible values:1,2,3 or a combination of them)`) – directions where splitting is allowed (for structured grids only)
- **type** (`integer`) – only for split by size (not resources): 0: centered splitting; 1: upwind splitting when better
- **R** (`integer`) – number of resources (processors)
- **minPtsPerDir** (`integer`) – minimum number of points per direction

Returns list of splitted grids

Return type [list of arrays] or [list of zones]

Example of use:

- Split a mesh by size (array):

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

a = G.cart((0,0,0),(1,1,1),(50,20,10))
B = T.splitSize(a, 2000, type=0)
C.convertArrays2File(B, 'out.plt')
```

- Split a mesh by size (pyTree):

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

a = G.cart((0,0,0),(1,1,1),(50,20,10))
t = C.newPyTree(['Base',a])
t = T.splitSize(t, 300, type=0)
C.convertPyTree2File(t, 'out.cgns')
```

Transform.**splitCurvatureAngle**(*a*, *sensibility*)

Split a curve defined by a 1D structured grid with respect to the curvature angle. If angle is lower than 180-sensibility in degrees or greater than 180+sensibility degrees, curve is split.

Parameters

- **a** (array or zone) – list of grids
- **sensibility** (float) – sensibility angle (in degrees) to allow splitting

Returns list of split curves

Return type [list of arrays] or [list of zones]

Example of use:

- Split a curve wrt curvature angle (array):

```
# - splitCurvatureAngle (array) -
import Converter as C
import Transform as T
import Geom as D

line = D.line((0.,0.,0.), (1.,1.,0.), 51)
line2 = D.line((-1.,1.,0.), (0.,0.,0.), 51)
a = T.join(line2, line)
list = T.splitCurvatureAngle(a, 30.)
C.convertArrays2File(list, 'out.plt')
```

- Split a curve wrt curvature angle (pyTree):

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

a = D.naca(12,101)
a2 = D.line((1,0,0), (2,0,0), 50)
a = T.join(a, a2)
a2 = D.line((2,0,0), (1,0,0), 50)
a = T.join(a, a2)
zones = T.splitCurvatureAngle(a, 20.)
C.convertPyTree2File(zones+[a], 'out.cgns')
```

Transform.**splitCurvatureRadius**(*a*, *Rs=100.*)

Split a curve defined by a 1D structured grid with respect to the curvature radius, using B-Spline approximations. The curve can be closed or not.

Parameters

- **a** (array or zone) – input mesh
- **Rs** (float) – threshold curvature radius below which the initial curve is split

Returns list of split curves

Return type [list of arrays] or [list of zones]

Example of use:

- Split a curve wrt curvature radius (array):

```
# - splitCurvatureRadius (array) -
import Converter as C
import Transform as T
import Geom as D

pts = C.array('x,y,z', 7, 1, 1)
x = pts[1][0]; y = pts[1][1]; z = pts[1][2]
x[0]= 6.; x[1] = 5.4; x[2]=4.8; x[3] = 2.5; x[4]  = 0.3
y[0]=10.; y[1]=0.036; y[2]=-5.;y[3]=0.21;y[4]=0.26;y[5]=7.
z[0]=1.; z[1]=1.; z[2]=1.;z[3]=1.;z[4]=1.;z[5]=1.; z[6]=1.

a = D.bezier( pts, 50 )
L = T.splitCurvatureRadius(a)
C.convertArrays2File([a]+L, 'out.plt')
```

- Split a curve wrt curvature radius (pyTree):

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

a = D.naca(12.5000)
zones = T.splitCurvatureRadius(a, 10.)
C.convertPyTree2File(zones+[a], 'out.cgns')
```

Transform.**splitConnexity**(*a*)

Split an unstructured mesh into connex parts.

Parameters **a** (array or zone) – input unstructured mesh

Returns list of connex parts

Return type [list of arrays] or [list of zones]

Example of use:

- Split an unstructured mesh into connex parts (array):

```
# - splitConnexity (array) -
import Converter as C
import Transform as T
import Geom as D

a = D.text2D("CASSIOPEE")
B = T.splitConnexity(a)
C.convertArrays2File(B, 'out.plt')
```

- Split an unstructured mesh into connex parts (pyTree):

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

a = D.text2D("CASSIOPEE")
B = T.splitConnexity(a)
C.convertPyTree2File(B, 'out.cgns')
```

Transform.**splitMultiplePts**(*a*, *dim=3*)

Split a structured mesh at external nodes connected to an even number of points, meaning that the geometrical point connects an odd number of blocks.

Parameters **a** ([list of arrays] or [list of zones]) – input set of structured grids

Returns set of structured grids after splitting

Return type [list of arrays] or [list of zones]

Example of use:

- Split a mesh at odd connections (array):

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

z0 = G.cart((0.,0.,0.),(0.1,0.1,1.),(10,10,1))
z1 = T.subzone(z0,(1,1,1),(5,10,1))
z2 = T.subzone(z0,(5,1,1),(10,5,1))
z3 = T.subzone(z0,(5,5,1),(10,10,1))
zones = [z1,z2,z3]
zones = T.splitMultiplePts(zones,dim=2)
C.convertArrays2File(zones, 'out.plt')
```

- Split a mesh at odd connections (pyTree):

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

(continues on next page)

(continued from previous page)

```
nk = 2
z0 = G.cart((0.,0.,0.),(0.1,0.1,1.),(10,10,nk))
z1 = T.subzone(z0,(1,1,1),(5,10,nk)); z1[0] = 'cart1'
z2 = T.subzone(z0,(5,1,1),(10,5,nk)); z2[0] = 'cart2'
z3 = T.subzone(z0,(5,5,1),(10,10,nk)); z3[0] = 'cart3'
z0 = T.translate(z0,(-0.9,0.,0.)); z0[0] = 'cart0'
z4 = G.cart((-0.9,0.9,0.),(0.1,0.1,1.),(19,5,nk)); z4[0] = 'cart4'
t = C.newPyTree(['Base',z1,z2,z3,z4])
t = X.connectMatch(t,dim=2)
t = C.fillEmptyBCWith(t, 'wall', 'BCWall', dim=2)
t = T.splitMultiplePts(t, dim=2)
C.convertPyTree2File(t, 'out.cgns')
```

Transform.**splitSharpEdges**(*a*, *alphaRef=30.*)

Split a 1D or 2D mesh at edges sharper than alphaRef. If the input grid is structured, then it returns an unstructured grid (BAR or QUAD).

Parameters

- **a** ([array, list of arrays] or [zone, list of zones]) – input mesh
- **alphaRef** (float) – angle (in degrees) below which the mesh must be split

Returns set of unstructured grids (with no sharp edges)

Return type [list of arrays] or [list of zones]

Example of use:

- Split a surface at sharp edges (array):

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

a = D.text3D("A"); a = G.close(a, 1.e-4)
B = T.splitSharpEdges(a, 89.)
C.convertArrays2File(B, 'out.plt')
```

- Split a surface at sharp edges (pyTree):

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

a = D.text3D("A"); a = G.close(a, 1.e-3)
B = T.splitSharpEdges(a, 89.)
C.convertPyTree2File(B, 'out.cgns')
```

Transform.**splitTBranches**(*a*, *tol=1.e-13*)

Split a curve defined by a 'BAR' if it has T-branches.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones]) – input mesh
- **tol** (float) – matching tolerance between points that define two branches

Returns set of BAR grids (with no T-branches)

Return type [list of arrays] or [list of zones]

Example of use:

- Split T-branches (array):

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

a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 10., (50,1,50))
c1 = T.subzone(a,(1,1,1),(50,1,1))
c2 = T.subzone(a,(1,1,50),(50,1,50))
c3 = T.subzone(a,(1,1,1),(1,1,50))
c = [c1,c2,c3]; c = C.convertArray2Hexa(c)
c = T.join(c)
res = T.splitTBranches(c)
C.convertArrays2File(res,"out.plt")
```

- Split T-branches (pyTree):

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

a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 10., (50,1,50))
c1 = T.subzone(a,(1,1,1),(50,1,1))
c2 = T.subzone(a,(1,50,1),(50,50,1))
c3 = T.subzone(a,(1,1,1),(1,50,1))
c = [c1,c2,c3]; c = C.convertArray2Hexa(c)
c = T.join(c)
res = T.splitTBranches(c)
C.convertPyTree2File(res, "out.cgns")
```

Transform.**splitManifold**(*a*)

Split a unstructured mesh (TRI or BAR only) into manifold pieces.

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

Returns set of TRI or BAR grids

Return type [list of arrays] or [list of zones]

Example of use:

- Split into manifold parts (array):

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

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

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

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

x = XOR.conformUnstr(s1, s2, 0., 2)
```

(continues on next page)

(continued from previous page)

```
x = T.splitManifold(x)

C.convertArrays2File(x, 'outS.plt')

a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 10., (50,1,50))
c1 = T.subzone(a,(1,1,1),(50,1,1))
c2 = T.subzone(a,(1,1,50),(50,1,50))
c3 = T.subzone(a,(1,1,1),(1,1,50))
c = [c1,c2,c3]; c = C.convertArray2Hexa(c)
c = T.join(c)
C.convertArrays2File([c], 'B.plt')
x = T.splitManifold(c)

C.convertArrays2File(x, 'outB.plt')
```

- Split into manifold parts (pyTree):

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

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

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

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

x = XOR.conformUnstr(s1, s2, 0., 2)
x = T.splitManifold(x)
C.convertPyTree2File(x, 'outS.cgns')

a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 10., (50,1,50))
c1 = T.subzone(a,(1,1,1),(50,1,1))
c2 = T.subzone(a,(1,50,1),(50,50,1))
c3 = T.subzone(a,(1,1,1),(1,50,1))
```

(continues on next page)

(continued from previous page)

```
c = [c1,c2,c3]; c = C.convertArray2Hexa(c)
c = T.join(c)
x = T.splitManifold(c)
C.convertPyTree2File(x, "outB.cgns")
```

Transform.**splitBAR**(*a*, *N*, *N2=-1*)

Split a curve defined by a BAR at index N. If N2 is provided, split also at index N2.

Parameters

- **a** (array or zone) – input mesh (BAR)
- **N** (integer) – index of split in a
- **N2** (integer) – optional second split index

Returns two BARS

Return type [list of arrays] or [list of zones]

Example of use:

- Split a BAR at given index (array):

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

a = G.cart((0,0,0), (1,1,1), (50,1,1))
a = C.convertArray2Tetra(a)
b = T.splitBAR(a, 5)
C.convertArrays2File(b, 'out.plt')
```

- Split a BAR at given index (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (50,1,1))
a = C.convertArray2Tetra(a)
B = T.splitBAR(a, 5)
C.convertPyTree2File(B, 'out.cgns')
```

Transform.**splitTRI**(*a*, *idxList*)

Split a triangular mesh into several triangular grids delineated by the polyline of indices idxList in the original TRI mesh.

Parameters

- **a** (array or zone) – input mesh (TRI)
- **idxList** (list of integers) – indices of split in a defining a polyline

Returns a set of TRI grids

Return type [list of arrays] or [list of zones]

Example of use:

- Split a TRI mesh with indices (array):

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

a = G.cart((0,0,0),(1,1,1),(5,5,1))
a = C.convertArray2Tetra(a)
C.convertArrays2File(a, 'out.plt')
c = [[10,16,22], [2,8,9]]
d = T.splitTRI(a, c)

C.convertArrays2File(d[0], 'out1.plt')
C.convertArrays2File(d[1], 'out2.plt')
```

- Split a TRI mesh with indices (pyTree):

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

a = D.circle( (0,0,0), 1, N=20 )
a = C.convertArray2Tetra(a)
a = G.close(a)
b = G.T3mesher2D(a)
c = [[9, 25, 27, 30, 29, 28, 34, 38, 0], [29, 23, 19, 20, 24, 29]]
```

(continues on next page)

(continued from previous page)

```
D = T.splitTRI(b, c)
C.convertPyTree2File(D, 'out.cgns')
```

3.5 Mesh deformation

Transform.**deform**(*a, vector=['dx','dy','dz']*)

Deform a surface by moving each point of a vector. The vector field must be defined in a, with the same location.

Exists also as an in-place version (_deform) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones]) – input surface mesh, containing the vector fields
- **vector** (list of 3 strings) – vector component names defined in a

Returns deformed surface mesh

Return type identical to input

Example of use:

- Deform a surface (array):

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

a = D.sphere((0,0,0), 1., 50)
n = G.getNormalMap(a)
n = C.center2Node(n); n[1] = n[1]*10
a = C.addVars([a,n])
b = T.deform(a,['sx','sy','sz'])
C.convertArrays2File([b], 'out.plt')
```

- Deform a surface (pyTree):

```
# - deform (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.),(10,10,10))
C._initVars(a,'dx', 10.)
C._initVars(a,'dy', 0)
C._initVars(a,'dz', 0)
b = T.deform(a)
C.convertPyTree2File(b, 'out.cgns')
```

Transform.**deformNormals**(*a*, *alpha*, *niter=1*)

Deform a surface mesh a by moving each point of the surface by a scalar field alpha times the surface normals in niter steps

Exists also as an in-place version (_deformNormals) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones]) – input surface mesh, containing the vector fields
- **alpha** (float) – factor of growth wrt to normals
- **niter** (integer) – number of steps (raise it to increase the smoothing of the resulting surface)

Returns deformed surface mesh

Return type identical to input

Example of use:

- Deform a surface following normals (array):

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

a = D.sphere((0,0,0), 1., 50)
a = C.convertArray2Hexa(a)
a = G.close(a)
b = C.initVars(a, '{alpha}=0.5*{x}')
b = C.extractVars(b, ['alpha'])
```

(continues on next page)

(continued from previous page)

```
b = T.deformNormals(a, b, niter=2)
C.convertArrays2File([b], 'out.plt')
```

- Deform a surface following normals (pyTree):

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

a = D.sphere6((0,0,0), 1., 10)
a = C.convertArray2Hexa(a)
a = T.join(a); a = G.close(a)

a = C.initVars(a, '{alpha}=0.5*{CoordinateX}')
a = T.deformNormals(a, 'alpha', niter=2)
C.convertPyTree2File(a, 'out.cgns')
```

Transform.**deformPoint**(*a*, *xyz*, *dxdydz*, *depth*, *width*)

Deform a surface mesh a by moving point P of vector V. Argument 'depth' controls the depth of deformation. Argument 'width' controls the width of deformation.

Exists also as an in-place version (_deformPoint) which modifies a and returns None.

Parameters

- **a** (`[array, list of arrays] or [zone, list of zones]`) – input surface mesh, containing the vector fields
- **P** (`3-tuple of floats`) – point that is moved
- **V** (`3-tuple of floats`) – vector of deformation
- **depth** (`float`) – to control the depth of deformation
- **width** (`float`) – to control the width of deformation

Returns deformed surface mesh

Return type identical to input

Example of use:

- Deform a surface at a point P (array):

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

a1 = G.cart((0,0,0), (1,1,1), (10,10,1))
a2 = T.deformPoint(a1, (0,0,0), (0.1,0.1,0.1), 0.5, 2.)
C.convertArrays2File([a2], "out.plt")
```

- Deform a surface at a point P (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (10,10,1))
a = T.deformPoint(a, (0,0,0), (0.1,0.1,1.), 0.5, 0.4)
C.convertPyTree2File(a, "out.cgns")
```

Transform.**deformMesh**(*a*, *surfDelta*, *beta=4.*, *type='nearest'*)

Deform a mesh defined by a given surface or a set of surfaces for which a deformation is defined at nodes as a vector field 'dx,dy,dz'. The surface surfDelta does not necessary match with a border of the meshes. Beta enables to extend the deformation region as multiplication factor of local deformation.

Exists also as an in-place version (_deformMesh) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones]) – input surface mesh, containing the vector fields
- **surfDelta** ([array,list of arrays] or [zone,list of zones]) – surface on which the deformation is defined

Returns deformed mesh

Return type identical to input

Example of use:

- Deform a mesh (array):

```
# - deformMesh (array) -
import Transform as T
import Converter as C
```

(continues on next page)

(continued from previous page)

```
import Geom as D

a1 = D.sphere6((0,0,0), 1, 20)
a1 = C.convertArray2Tetra(a1); a1 = T.join(a1)
point = C.getValue(a1, 0)
a2 = T.deformPoint(a1, point, (0.1,0.05,0.2), 0.5, 2.)
delta = C.addVars(a1, ['dx','dy','dz'])
delta = C.extractVars(delta, ['dx','dy','dz'])
delta[1][:,:] = a2[1][:,:]-a1[1][:,:]
a1 = C.addVars([a1, delta])
m = D.sphere6((0,0,0), 2, 20)
m = T.deformMesh(m, a1)
C.convertArrays2File(m, "out.plt")
```

- Deform a mesh (pyTree):

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

a1 = D.sphere6((0,0,0),1,20)
a1 = C.convertArray2Tetra(a1); a1 = T.join(a1)
point = C.getValue(a1, 'GridCoordinates', 0)
a2 = T.deformPoint(a1, point, (0.1,0.05,0.2), 0.5, 2.)
delta = C.diffArrays(a2,a1)
deltax = C.getField('DCoordinateX',delta)
deltay = C.getField('DCoordinateY',delta)
deltaz = C.getField('DCoordinateZ',delta)
for noz in range(len(deltax)):
    deltax[noz][0] = 'dx'
    deltay[noz][0] = 'dy'
    deltaz[noz][0] = 'dz'
a1 = C.setFields(deltax,a1,'nodes')
a1 = C.setFields(deltay,a1,'nodes')
a1 = C.setFields(deltaz,a1,'nodes')

m = D.sphere6((0,0,0),2,20)
m = T.deformMesh(m, a1)
C.convertPyTree2File(m, "out.cgns")
```

3.6 Mesh projections

Transform.**projectAllDirs**(*a, s, vect=['nx','ny','nz'], oriented=0*)

Project a surface mesh a onto a set of surfaces s according to a vector defined for each point of the mesh a. If oriented=0, both directions are used for projection, else the vector direction is used.

Exists also as an in-place version (_projectAllDirs) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones]) – input surface mesh, containing the vector fields
- **s** ([array,list of arrays] or [zone,list of zones]) – projection surface
- **vect** (list of 3 strings) – vector component names
- **oriented** (integer (0 or 1)) – 0 for projection in the vector direction and also in its opposite direction

Returns projected mesh

Return type identical to input

Example of use:

- Project a mesh (array):

```
# - projectAllDirs (array) -
import Geom as D
import Converter as C
import Generator as G
import Transform as T
a = D.sphere6((0,0,0), 1., 20)
b = G.cart((1.1,-0.1,-0.1),(0.03,0.03,0.03), (1,50,50))
n = G.getNormalMap(b)
n = C.center2Node(n)
b = C.addVars([b,n])
c = T.projectAllDirs([b], a, ['sx','sy','sz'])
C.convertArrays2File([b]+c, 'out.plt')
```

- Project a mesh (pyTree):

```
# - projectAllDirs (pyTree) -
import Geom.PyTree as D
```

(continues on next page)

(continued from previous page)

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

a = D.sphere((0,0,0), 1., 20)
b = G.cart((1.1,-0.1,-0.1),(0.1,0.1,0.1), (1,5,5))
b = G.getNormalMap(b)
b = C.center2Node(b,['centers:sx','centers:sy','centers:sz'])
c = T.projectAllDirs(b, a,['sx','sy','sz']); c[0] = 'projection'
C.convertPyTree2File([a,b,c], 'out.cgns')
```

Transform.**projectDir**(*a*, *s*, *dir*, *smooth=0*, *oriented=0*)

Project a surface mesh a onto a set of surfaces s following a constant direction dir. If oriented=0, both directions are used for projection, else the vector direction is used. If smooth=1, points that cannot be projected are smoothed (available only for structured grids).

Exists also as an in-place version (_projectDir) which modifies a and returns None.

Parameters

- **a** (`[array list of arrays] or [zone, list of zones]`) – input surface mesh
- **s** (`[array, list of arrays] or [zone, list of zones]`) – projection surface
- **dir** (`3-tuple of floats`) – constant vector that directs the projection
- **smooth** (`integer (0 or 1)`) – smoothing of unprojected points
- **oriented** (`integer (0 or 1)`) – 0 for projection in the vector direction and also in its opposite direction

Returns projected mesh

Return type identical to input

Example of use:

- Project a mesh following a direction (array):

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

(continues on next page)

(continued from previous page)

```
a = D.sphere((0,0,0), 1., 20)
b = G.cart((1.1,-0.1,-0.1),(0.03,0.03,0.03), (1,50,50))
c = T.projectDir(b, [a], (1.,0,0))
d = T.projectDir([b], [a], (1.,0,0), smooth=1)
C.convertArrays2File([a,b,c]+d, 'out.plt')
```

- Project a mesh following a direction (pyTree):

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

a = D.sphere((0,0,0), 1., 20)
b = G.cart((1.1,-0.1,-0.1),(0.1,0.1,0.1), (1,5,5))
c = T.projectDir(b, a, (1.,0,0)); c[0] = 'projection'
C.convertPyTree2File([a,b,c], 'out.cgns')
```

Transform.**projectOrtho**(*a*, *s*)

Project a surface mesh a orthogonally onto a set of surfaces s.

Exists also as an in-place version (_projectOrtho) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones]) – input surface mesh, containing the vector fields
- **s** ([array, list of arrays] or [zone, list of zones]) – projection surface

Returns projected mesh

Return type identical to input

Example of use:

- Project a mesh orthogonally (array):

```
# - projectOrtho (array) -
import Geom as D
import Converter as C
import Generator as G
import Transform as T
a = D.sphere((0,0,0), 1., 300)
```

(continues on next page)

(continued from previous page)

```
b = G.cart((-0.5,-0.5,-1.5),(0.05,0.05,0.1), (20,20,1))
c = T.projectOrtho(b, [a])
C.convertArrays2File([a,b,c], 'out.plt')
```

- Project a mesh orthogonally (pyTree):

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

a = D.sphere((0,0,0), 1., 20)
b = G.cart((1.1,-0.1,-0.1),(0.1,0.1,0.1), (1,5,5))
c = T.projectOrtho(b, a); c[0] = 'projection'
C.convertPyTree2File([a,b,c], 'out.cgns')
```

Transform.**projectOrthoSmooth**(*a*, *s*, *niter=1*)

Project a surface mesh a following smoothed normals onto a set of surfaces s.

Exists also as an in-place version (_projectOrthoSmooth) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones]) – input surface mesh
- **s** ([array, list of arrays] or [zone, list of zones]) – projection surface
- **niter** (integer) – number of smoothing iterations

Returns projected mesh

Return type identical to input

Example of use:

- Project a mesh following smoothed normals (array):

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

(continues on next page)

(continued from previous page)

```
a = D.sphere((0,0,0), 1., 30)
b = G.cart((-0.5,-0.5,-1.5),(0.05,0.05,0.1), (20,20,1))
c = T.projectOrthoSmooth(b, [a], niter=2)
C.convertArrays2File([a,b,c], 'out.plt')
```

- Project a mesh following smoothed normals (pyTree):

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

a = D.sphere((0,0,0), 1., 30)
b = G.cart((-0.5,-0.5,-1.5),(0.05,0.05,0.1), (20,20,1))
c = T.projectOrthoSmooth(b, [a], niter=2)
C.convertPyTree2File([a,b,c], 'out.cgns')
```

Transform.**projectRay**(*a*, *s*, *P*)

Project a surface mesh a onto a set of surfaces s following rays starting from a point P.

Exists also as an in-place version (_projectRay) which modifies a and returns None.

Parameters

- **a** ([array, list of arrays] or [zone, list of zones]) – input surface mesh
- **s** ([array, list of arrays] or [zone, list of zones]) – projection surface
- **P** (3-tuple of floats) – starting point of rays

Returns projected mesh

Return type identical to input

Example of use:

- Project a mesh following rays (array):

```
# - projectRay (array) -
import Geom as D
import Converter as C
```

(continues on next page)

(continued from previous page)

```
import Generator as G
import Transform as T
a = D.sphere((0,0,0), 1., 20)
b = G.cart((1.1,-0.1,-0.1),(0.1,0.1,0.1), (1,5,5))
c = T.projectRay(b, a, (0,0,0))
C.convertArrays2File([a,b,c], 'out.plt')
```

- Project a mesh following rays (pyTree):

```
# - projectRay (pyTree) -
import Geom.PyTree as D
import Converter.PyTree as C
import Generator.PyTree as G
import Transform.PyTree as T
a = D.sphere((0,0,0), 1., 20)
b = G.cart((1.1,-0.1,-0.1),(0.1,0.1,0.1), (1,5,5))
c = T.projectRay(b, a, (0,0,0)); c[0] = 'projection'
C.convertPyTree2File([a,b,c], 'out.cgns')
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search