Connector Documentation

Release 3.5

/ELSA/MU-090XX/V3.5

Nov 21, 2022

CONTENTS

CHAPTER

ONE

PREAMBLE

Connector module is used to compute connectivity between meshes. It manipulates arrays (as defined in Converter documentation) or CGNS/Python trees (pyTrees) as data structures.

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

To use the Connector module with the array interface:

```
import Connector as X
```

With the pyTree interface:

```
import Connector.PyTree as X
```

CHAPTER

TWO

LIST OF FUNCTIONS

– Multiblock connectivity

Connector.connectMatch(a1, a2[, sameZone, ...])	Find matching boundaries.
Connector.PyTree.connectMatchPeriodic(t[, ...])	Find periodic matching boundaries.
Connector.PyTree.connectNearMatch(t[, ...])	Find boundaries that matches with a given ratio.
Connector.PyTree.setDegeneratedBC(t[, dim, tol])	Find degenerated boundaries (lines).

– Overset grid connectivity

Connector.blankCells(coords, cellnfields, body)	Blank cells in coords by a X-Ray mask defined by the body, within a distance delta.
Connector.blankCellsTetra(coords, ...[, ...])	Blank cells in coords (by setting the cellN to cellnval) falling inside a Tetra Mesh mask defined by meshT4.
Connector.blankCellsTri(coords, cellnfields, ...)	Blank cells in coords (by setting the cellN to cellnval) falling inside a Triangular surface mesh mask defined by meshT3.
Connector.blankIntersectingCells(a, cellN[, tol])	Blank intersecting cells in a zone.
Connector.setHoleInterpolatedPoints(cellN, ...])	Set interpolated points cellN=2 around cellN=0 points.
Connector.optimizeOverlap(nodes1, centers1, ...)	Optimize the overlap of grids defined by nodes1 and nodes2 centers1 and centers2 define the coordinates of cell centers, modified by the double wall algorithm + cellN variable.

Continued on next page

Table 2 – continued from previous page

Connector.maximizeBlankedCells(a[, depth, ...])	Maximize the blanked region.
Connector.PyTree.cellN2OversetHoles(t[, append])	Create OversetHole nodes from cellN field.
Connector.PyTree.setInterpData(tR, tD[, ...])	Compute and store overset interpolation data.
Connector.PyTree.getOversetInfo(aR, topTreeD)	Return information on overset connectivities.
Connector.PyTree.extractChimeraInfo(a[, ...])	Extract interpolated/extrapolated/orphan points as zones.

– Overset grid connectivity for elsA solver

Connector.PyTree.setInterpolations(t[, loc, ...])	Compute interpolation data for chimera and for elsA solver.
Connector.PyTree.chimeraInfo(a[, type])	Extract Overset information when computed with setInterpolations.

CHAPTER

THREE

CONTENTS

3.1 Multiblock connectivity

Connector.**connectMatch**()

Detect and set all matching windows, even partially.

Using the array interface:

```
res = X.connectMatch(a1, a2, sameZone=0, tol=1.e-6, dim=3)
```

Detect and set all matching windows between two structured arrays a1 and a2. Return the subrange of indices of abutting windows and an index transformation from a1 to a2. If the CFD problem is 2D, then dim must be set to 2. Parameter sameZone must be set to 1 if a1 and a2 define the same zone.

Parameters **a1,a2** (arrays) – Input data

Using the PyTree interface:

```
t = X.connectMatch(t, tol=1.e-6, dim=3)
```

Detect and set all matching windows in a zone node, a list of zone nodes or a complete pyTree. Set automatically the Transform node corresponding to the transformation from matching block 1 to block 2. If the CFD problem is 2D, then dim must be set to 2.

Parameters **t** (pyTree, base, zone, list of zones) – input data

Return type identical to input

Exists also as parallel distributed version (X.Mpi.connectMatch).

Example of use:

- Detect matching boundaries of a mesh (array):

```
# - connectMatch (array) -
import Generator as G
import Connector as X
import Geom as D
import Transform as T
import Converter as C
# 3D raccord i = 1 partiel profil NACA
msh = D.naca(12., 5001)
msh2 = D.line((1.,0.,0.),(2.,0.,0.),5001); msh = T.join(msh, msh2)
msh2 = D.line((2.,0.,0.),(1.,0.,0.),5001); msh = T.join(msh2, msh)
Ni = 300; Nj = 50
distrib = G.cart((0,0,0), (1./(Ni-1), 0.5/(Nj-1),1), (Ni,Nj,1))
naca = G.hyper2D(msh, distrib, "C")
res = X.connectMatch(naca,naca,sameZone=1,dim=2)
C.convertArrays2File([naca],"out.plt")
print(res)
```

- Add 1-to-1 abutting connectivity in a pyTree (pyTree):

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

a1 = G.cart((0.,0.,0.), (0.1, 0.1, 0.1), (11, 21, 3)); a1[0] = 'cart1'
a2 = G.cart((1., 0.2, 0.), (0.1, 0.1, 0.1), (11, 21, 3)); a2[0] = 'cart2'
t = C.newPyTree(['Base',a1,a2])
t = X.connectMatch(t)
C.convertPyTree2File(t, 'out.cgns')
```

Connector.PyTree.**connectMatchPeriodic**(*t, rotationCenter=[0.,0.,0.], rotationAngle=[0.,0.,0.], translation=[0.,0.,0.], tol=1.e-6, dim=3, unitAngle=None*)

Detect and set all periodic matching borders, even partially, in a zone node, a list of zone nodes, a base, or a full pyTree. Periodicity can be defined either by rotation or translation or by a composition of rotation and translation.

Parameters **t** (pyTree, base, zone, list of zones) – input data

Return type identical to input

Set automatically the Transform node corresponding to the transformation from matching block 1 to block 2, and the 'GridConnectivityProperty/Periodic' for periodic matching BCs.

If the CFD problem is 2D, then dim must be set to 2.

For periodicity by rotation, the rotation angle units can be specified by argument unitAngle, which can be 'Degree','Radian',None.

If unitAngle=None or 'Degree': parameter rotationAngle is assumed to be defined in degrees.

If unitAngle='Radian': parameter rotationAngle is assumed in radians.

Exists also as parallel distributed version (X.Mpi.connectMatchPeriodic).

Note:

- if the mesh is periodic in rotation and in translation separately (i.e. connecting with some blocks in rotation, and some other blocks in translation), the function must be applied twice.
- Since *Cassiopee2.6*: 'RotationAngle' node in 'Periodic' node is always defined in Radians. A DimensionalUnits child node is also defined.

Example of use:

- Add periodic 1-to-1 abutting grid connectivity in a pyTree (pyTree):

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

a = G.cylinder((0.,0.,0.), 0.1, 1., 0., 90., 5., (11,11,11))
t = C.newPyTree(['Base',a])
t = X.connectMatchPeriodic(t, rotationCenter=[0.,0.,0.],
                           translation=[0.,0.,5.])
t = X.connectMatchPeriodic(t, rotationCenter=[0.,0.,0.],
                           rotationAngle=[0.,0.,90.])
C.convertPyTree2File(t, 'out.cgns')
```

Connector.PyTree.**connectNearMatch**(*t*, *ratio=2*, *tol=1.e-6*, *dim=3*)

Detect and set all near-matching windows, even partially in a zone node, a list of zone nodes or a complete pyTree. A 'UserDefinedData' node is set, with the PointRangeDonor, the Transform and NMRatio nodes providing information for the opposite zone. .. warning:: connectMatch must be applied first if matching windows exist.

Parameter ratio defines the ratio between near-matching windows and can be an integer (e.g. 2) or a list of 3 integers (e.g. [1,2,1]), specifying the nearmatching direction to test (less CPU-consuming). If the CFD problem is 2D, then dim must be set to 2.

Parameters **t** (pyTree, base, zone, list of zones) – input data

Return type identical to input

Example of use:

- Add n-to-m abutting grid connectivity in a pyTree (pyTree):

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

a1 = G.cart((0.,0.,0.), (0.1, 0.1, 0.1), (11, 21, 3)); a1[0] = 'cart1'
a2 = G.cart((1., 0.2, 0.), (0.1, 0.1, 0.1), (11, 21, 3)); a2[0] = 'cart2'
a2 = T.oneovern(a2,(1,2,1))
t = C.newPyTree(['Base',a1,a2])
t = X.connectNearMatch(t)
C.convertPyTree2File(t, 'out.cgns')
```

Connector.PyTree.**setDegeneratedBC**(*t*, *dim=3*, *tol=1.e-10*)

Detect all degenerated lines in 3D zones and define a BC as a 'BCDegenerateLine' BC type. For 2D zones, 'BCDegeneratePoint' type is defined. If the problem is 2D according to (i,j), then parameter 'dim' must be set to 2. Parameter 'tol' defines a distance below which a window is assumed degenerated.

Example of use:

- Add degenerated line as BCs in a pyTree (pyTree):

```
# - setDegeneratedBC (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X
a = G.cylinder((0,0,0), 0., 1., 360., 0., 1, (21,21,21))
t = C.newPyTree(['Base',a])
t = X.setDegeneratedBC(t)
C.convertPyTree2File(t, "out.cgns")
```

3.2 Overset connectivity

Connector.**blankCells**()

Blank cells using X-Ray method.

Using the array interface:

```
cellns = X.blankCells(coords, cellns, body, blankingType=2, delta=1.e-
↪10, dim=3, masknot=0, tol=1.e-8)
```

Blank the cells of a list of grids defined by coords (located at nodes). The X-Ray mask is defined by bodies, which is a list of arrays. Cellnaturefield defined in cellns is modified (0: blanked points, 1: otherwise). Some parameters can be specified: blankingType, delta, masknot, tol. Their meanings are described in the table below:

Parameter value	Meaning
blankingType=0	blank nodes inside bodies (node_in).
blankingType=2	blank cell centers inside bodies (center_in).
blankingType=1	blank cell centers intersecting with body (cell_intersect).
blankingType=2	blank cell centers using an optimized cell intersection (cell_intersect_opt) and interpolation depth=2 (blanking region may be reduced where blanking point can be interpolated).
blankingType=1	blank cell centers using an optimized cell intersection (cell_intersect_opt) and interpolation depth=1.
delta=0.	cells are blanked in the body
delta greater than 0.	the maximum distance to body, in which cells are blanked
masknot=0	Classical blanking applied
masknot=1	Inverted blanking applied: cells out of the body are blanked
dim=3	body described by a surface and blanks 3D cells.
dim=2	body blanks 2D or 1D zones.
tol=1.e-8 (default)	tolerance for the multiple definition of the body.

Note: in case of blankingType=0, location of cellns and coords must be identical.

Using the pyTree interface:

```
B = X.blankCells(t, bodies, BM, depth=2, blankingType='cell_intersect',␣
↪delta=1.e-10, dim=3, tol=1.e-8, XRaydim1=1000, XRaydim2=1000)
```

blankCells function sets the cellN to 0 to blanked nodes or cell centers of both structured and unstructured grids.

The location of the cellN field depends on the *blankingType* parameter: if 'node_in' is used, nodes are blanked, else centers are blanked.

The mesh to be blanked is defined by a pyTree t, where each basis defines a Chimera component. The list of bodies blanking the grids is defined in bodies.

Each element of the list bodies is a set of CGNS/Python zones defining a closed and watertight surface.

The blanking matrix BM is a numpy array of size nbases x nbodies.

BM(i,j)=1 means that ith basis is blanked by jth body.

BM(i,j)=0 means no blanking, and BM(i,j)=-1 means that inverted hole-cutting is performed.

blankingType can be 'cell_intersect', 'cell_intersect_opt', 'center_in' or 'node_in'. Parameter depth is only meaningfull for 'cell_intersect_opt'.

XRaydim1 and XRaydim2 are the dimensions of the X-Ray hole-cutting in the x and y directions in 3D.

If the variable 'cellN' does not exist in the input pyTree, it is initialized to 1, located at 'nodes' if 'node_in' is set, and at centers in other cases.

Warning: 'cell_intersect_opt' can be CPU time-consuming when delta>0.

Example of use:

- Blank cells (array):

```
# - blankCells (array) -
import Converter as C
import Connector as X
import Generator as G
```

(continues on next page)

(continued from previous page)

```
import Geom as D

surf = D.sphere((0,0,0), 0.5, 20)
surf = C.convertArray2Tetra(surf)

a = G.cart((-1.,-1.,-1.),(0.1,0.1,0.1), (20,20,20))
ca = C.array('cellN',19,19,19)
ca = C.initVars(ca, 'cellN', 1.)
celln = X.blankCells([a], [ca], [surf], blankingType=1, delta=0.)
a = C.node2Center(a)
celln = C.addVars([[a], celln])
C.convertArrays2File(celln, 'out.plt')

# in place-modifies cellN
surf = D.sphere((0,0,0), 0.5, 20)
surf = C.convertArray2Tetra(surf)

a = G.cart((-1.,-1.,-1.),(0.1,0.1,0.1), (20,20,20))
ca = C.array('cellN',19,19,19)
ca = C.initVars(ca, 'cellN', 1.)
X._blankCells([a], [ca], [surf], blankingType=1, delta=0.)
a = C.node2Center(a)
celln = C.addVars([[a], celln])
C.convertArrays2File(celln, 'out.plt')
```

- Blank cells (pyTree):

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

surf = D.sphere((0,0,0), 0.5, 20)

a = G.cart((-1.,-1.,-1.),(0.1,0.1,0.1), (20,20,20))
t = C.newPyTree(['Cart',a])
C._initVars(t, 'centers:cellN', 1.)

bodies = [[surf]]
# Matrice de masquage (arbre d'assemblage)
import numpy
BM = numpy.array([ [ 1] ] )
```

(continues on next page)

(continued from previous page)

```
t = X.blankCells(t, bodies, BM, blankingType='cell_intersect', delta=0.)
C.convertPyTree2File(t, 'out.cgns')
```

Connector.**blankCellsTetra**()

Using the array interface:

```
cellns = X.blankCellsTetra(coords, cellns, body, blankingType=2,␣
↪tol=1.e-12)
```

Blanks the input grids nodes or cells (depending on the *blankingType* value) that fall inside a volume body mask. The blanking is achieved by setting the Cellnaturefield to *cellnval* (0 by default) in *cellns*.

The input grids are defined by coords located at nodes as a list of arrays. The body mask is defined by sets of tetrahedra in any orientation, as a list of arrays.

If the *blankingMode* is set to 1 (overwrite mode), Cellnaturefield is reset to 1 for any node/cell outside the body mask. Hence the value of 1 is forbidden for cellnval upon entry (it will be replaced by 0).

The parameters meanings and values are described in the table below:

Parameter value	Meaning
blanking-Type=0	blanks the nodes falling inside the body masks (node_in).
blanking-Type=2	blanks the cells having their center falling inside the body masks (center_in).
blanking-Type=1	blanks the cells that intersect or fall inside the body masks (cell_intersect).
tol=1.e-12	tolerance for detecting intersections (NOT USED CURRENTLY).
cellnval=0 (default)	value used for flagging as blanked.
blanking-Mode=0 (default)	Appending mode: cellns is only modified for nodes/cells falling inside the body mask by setting the value in cellns to cellnval.
blanking-Mode=1	Overwriting mode: cellns is modified for both nodes/cells falling inside (set to cellnval) and outside (set to 1) the body mask.

> **Warning:** in case of blankingType=0, location of cellns and coords must be identical.

Using the pyTree interface:

```
B = X.blankCellsTetra(t, bodies, BM, blankingType='node_in', tol=1.e-
↪12, cellnval=0, overwrite=0)
```

Blanks the input grids nodes or cells (depending on the *blankingType* value) that fall inside a volume body mask.

The blanking is achieved by setting the Cellnaturefield to *cellnval* (0 by default) in *cellns*.

The mesh to be blanked is defined by a pyTree t, where each basis defines a Chimera component. The list of bodies blanking the grids is defined in bodies.

Each element of the list bodies is a set of CGNS/Python zones defining a tetrahedra mesh.

The blanking matrix BM is a numpy array of size nbases x nbodies.

BM(i,j)=1 means that ith basis is blanked by jth body.

BM(i,j)=0 means no blanking, and BM(i,j)=-1 means that inverted hole-cutting is performed.

blankingType can be 'cell_intersect', 'center_in' or 'node_in'.

If the variable 'cellN' does not exist in the input pyTree, it is initialized to 1, located at 'nodes' if 'node_in' is set, and at centers in other cases.

If the *overwrite* is set to 1 (overwrite mode), Cellnaturefield is reset to 1 for any node/cell outside the body mask.

Hence the value of 1 is forbidden for cellnval upon entry (it will be replaced by 0).

Example of use:

- Blank cells with a tetra mesh (array):

```
# - blankCellsTetra (array) - 'NODE IN'
import Converter as C
import Connector as X
import Generator as G

# Tet mask
```

(continues on next page)

(continued from previous page)

```
mT4 = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
mT4 = C.convertArray2Tetra(mT4)
#C.convertArrays2File([mT4], 'maskT4.plt', 'bin_tp')

# Mesh to blank
a = G.cart((-5.,-5.,-5.), (0.5,0.5,0.5), (100,100,100))
#C.convertArrays2File([a], 'bgm.plt')

ca = C.array('cellN',100,100,100)
ca = C.initVars(ca, 'cellN', 1.)
#C.convertArrays2File([ca], 'ca.plt')

celln = X.blankCellsTetra([a], [ca], [mT4], blankingType=0, tol=1.e-12)

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

- Blank cells with a tetra mesh (pyTree):

```
# - blankCellsTetra (pyTree) - 'NODE IN'
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G

# Tet mask
mT4 = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
mT4 = C.convertArray2Tetra(mT4)

# Mesh to blank
a = G.cart((-5.,-5.,-5.), (0.5,0.5,0.5), (100,100,100))

t = C.newPyTree(['Cart',a])
C._initVars(t, 'centers:cellN', 1.)

masks = [[mT4]]
# Matrice de masquage (arbre d'assemblage)
import numpy
BM = numpy.array([[1]])

t1 = X.blankCellsTetra(t, masks, BM, blankingType='node_in', tol=1.e-12)
C.convertPyTree2File(t1, 'out.cgns')

t2 = C.convertArray2Tetra(t)
t2 = X.blankCellsTetra(t2, masks, BM, blankingType='node_in', tol=1.e-12)
```

(continues on next page)

(continued from previous page)

```
C.convertPyTree2File(t2, 'out2.cgns')

t3 = C.convertArray2NGon(t)
t3 = X.blankCellsTetra(t3, masks, BM, blankingType='node_in', tol=1.e-12)
C.convertPyTree2File(t3, 'out3.cgns')
```

Connector.**blankCellsTri**()

Using the array interface:

```
cellns = X.blankCellsTri(coords, cellns, body, blankingType=2, tol=1.
↪e-12, cellnval=0, blankingMode=0)
```

Blanks the input grids nodes or cells (depending on the *blankingType* value) that fall inside a surfacic body mask.

The blanking is achieved by setting the Cellnaturefield to *cellnval* (0 by default) in *cellns*.

The input grids are defined by coords located at nodes as a list of arrays. The body mask is defined by triangular surface meshes in any orientation, as a list of arrays.

If the *blankingMode* is set to 1 (overwrite mode), Cellnaturefield is reset to 1 for any node/cell outside the body mask. Hence the value of 1 is forbidden for cellnval upon entry (it will be replaced by 0).

The parameters meanings and values are described in the table below:

Parameter value	Meaning
blankingType=0	blanks the nodes falling inside the body masks (node_in).
blankingType=2	blanks the cells having their center falling inside the body masks (center_in).
blankingType=1	blanks the cells that intersect or fall inside the body masks (cell_intersect).
tol=1.e-12 (default)	tolerance for detecting intersections (NOT USED CURRENTLY).
cellnval=0 (default)	value used for flagging as blanked.
blankingMode=0 (default)	Appending mode: cellns is only modified for nodes/cells falling inside the body mask by setting the value in cellns to cellnval.
blankingMode=1	Overwriting mode: cellns is modified for both nodes/cells falling inside (set to cellnval) and outside (set to 1) the body mask.

Warning: in case of blankingType=0, location of cellns and coords must be identical.

Using the pyTree interface:

```
B = X.blankCellsTri(t, bodies, BM, blankingType='node_in', tol=1.e-12,
↪ cellnval=0, overwrite=0)
```

Blanks the input grids nodes or cells (depending on the *blankingType* value) that fall inside a volume body mask.

The blanking is achieved by setting the Cellnaturefield to *cellnval* (0 by default) in *cellns*.

The mesh to be blanked is defined by a pyTree t, where each basis defines a Chimera component. The list of bodies blanking the grids is defined in bodies.

Each element of the list bodies is a set of CGNS/Python zones defining a triangular watertight closed surface.

The blanking matrix BM is a numpy array of size nbases x nbodies.

BM(i,j)=1 means that ith basis is blanked by jth body.

BM(i,j)=0 means no blanking, and BM(i,j)=-1 means that inverted hole-cutting is performed.

blankingType can be 'cell_intersect', 'center_in' or 'node_in'.

If the variable 'cellN' does not exist in the input pyTree, it is initialized to 1, located at 'nodes' if 'node_in' is set, and at centers in other cases.

If the *overwrite* is set to 1 (overwrite mode), Cellnaturefield is reset to 1 for any node/cell outside the body mask.

Hence the value of 1 is forbidden for cellnval upon entry (it will be replaced by 0).

Example of use:

- Blank cells with a triangular surface mask (array):

```
# - blankCellsTri (array) - 'NODE IN'
import Converter as C
import Connector as X
import Generator as G
import Geom as D
import Post as P

# Tri mask
m = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
m = P.exteriorFaces(m)

# Mesh to blank
a = G.cart((-5.,-5.,-5.), (0.5,0.5,0.5), (100,100,100))
# celln init
ca = C.array('cellN',100,100,100)
ca = C.initVars(ca, 'cellN', 1.)
# Blanking
celln = X.blankCellsTri([a], [ca], m, blankingType=0, tol=1.e-12)
celln = C.addVars([[a], celln])
C.convertArrays2File(celln, 'out0.plt')
```

- Blank cells with a triangular surface mask (pyTree):

```
# - blankCellsTri (pyTree) - 'NODE IN'
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G
import Geom.PyTree as D
import Post.PyTree as P
```

(continues on next page)

(continued from previous page)

```
# Tet mask
m = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,10,10))
m = P.exteriorFaces(m)

# Mesh to blank
a = G.cart((-5.,-5.,-5.), (0.5,0.5,0.5), (100,100,100))

t = C.newPyTree(['Cart',a])
C._initVars(t, 'centers:cellN', 1.)

masks = [[m]]
# Matrice de masquage (arbre d'assemblage)
import numpy
BM = numpy.array([[1]])

t = X.blankCellsTri(t, masks, BM, blankingType='node_in', tol=1.e-12)
C.convertPyTree2File(t, 'out.cgns')
```

Connector.**setHoleInterpolatedPoints**()

Using the array interface:

```
a = X.setHoleInterpolatedPoints(a, depth=2, dir=0, loc='centers',␣
↪cellNName='cellN')
```

Compute the fringe of interpolated points around a set of blanked points in a mesh a. Parameter depth is the number of layers of interpolated points to be set. If depth > 0 the fringe of interpolated points is set outside the blanked zones, whereas if depth < 0, the depth layers of blanked points are marked as to be interpolated. If dir=0, uses a directional stencil of depth points, if dir=1, uses a star shape stencil, if dir=2, uses a diamond stencil. Blanked points are identified by the variable 'cellN'; 'cellN' is set to 2 for the fringe of interpolated points. If cellN is located at cell centers, set loc parameter to 'centers', else loc='nodes'.

Using the pyTree interface:

```
t = X.setHoleInterpolatedPoints(t, depth=2, dir=0, loc='centers',␣
↪cellNName='cellN')
```

Compute the fringe of interpolated points around a set of blanked points in a pyTree t. Parameter depth is the number of layers of interpolated points that are built; if depth > 0 the fringe of interpolated points is outside the blanked zones, and if depth < 0, it is built towards the inside. If dir=0, uses a directional stencil of depth points, if dir=1, uses a star shape stencil, if dir=2, uses a diamond stencil. Blanked points are identified by the variable

'cellN' located at mesh nodes or centers. 'cellN' is set to 2 for the fringe of interpolated points.

Example of use:

- Set the fringe of interpolated points near blanked points (array):

```
# - setHoleInterpolatedPts (array) -
import Converter as C
import Connector as X
import Generator as G

def sphere(x,y,z):
    if x*x+y*y+z*z < 0.5**2 : return 0.
    else: return 1.

a = G.cart((-1.,-1.,-1.),(0.1,0.1,0.1), (20,20,20))
celln = C.node2Center(a)
celln = C.initVars(celln, 'cellN', sphere, ['x','y','z'])
celln = X.setHoleInterpolatedPoints(celln, depth=1)
C.convertArrays2File([celln], 'out.plt')
```

- Set the fringe of interpolated points near the blanked points (pyTree):

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

N = 101
h = 2./(N-1)
a = G.cart((-1.,-1.,-1.),(h,h,h), (N,N,N))
t = C.newPyTree(['Cart', a])
C._initVars(t,'{centers:cellN}=(1.-({centers:CoordinateX}*{centers:CoordinateX}+
↪{centers:CoordinateY}*{centers:CoordinateY}+{centers:CoordinateZ}*
↪{centers:CoordinateZ}<0.25))')

X._setHoleInterpolatedPoints(t, depth=1)
C.convertPyTree2File(t, 'out.cgns')
```

Connector.**optimizeOverlap**()

Using the array interface:

```
cellns = X.optimizeOverlap(nodes1, centers1, nodes2, centers2,␣
↪prio1=0, prio2=0)
```

Optimize the overlap between two zones defined by nodes1 and nodes2, centers1 and centers2 correspond to the mesh located at centers and the field 'cellN'. The field 'cellN' located at centers is set to 2 for interpolable points. Priorities can be defined for zones: prio1=0 means that the priority of zone 1 is high. If two zones have the same priority, then the cell volume criterion is used to set the cellN to 2 for one of the overlapping cells, the other not being modified. If the priorities are not specified, the cell volume criterion is applied also:

Using the pyTree interface:

```
t = X.optimizeOverlap(t, double_wall=0, priorities=[], planarTol=0.)
```

Optimize the overlapping between all structured zones defined in a pyTree t. The 'cellN' variable located at cell centers is modified, such that cellN=2 for a cell interpolable from another zone. Double wall projection technique is activated if 'double_wall'=1. Parameter planarTol can be useful for double wall cases, in the case when double wall surfaces are planar but distant from planarTol to each other. The overlapping is optimized between zones from separated bases, and is based on a priority to the cell of smallest size. One can impose a priority to a base over another base, using the list priorities. For instance, priorities = ['baseName1',0, 'baseName2',1] means that zones from base of name 'baseName1' are preferred over zones from base of name 'baseName2':

Example of use:

- Optimize overlapping (array):

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

Ni = 50; Nj = 50; Nk = 2
a = G.cart((0,0,0),(1./(Ni-1), 1./(Nj-1),1), (Ni,Nj,Nk))
b = G.cart((0,0,0),(2./(Ni-1), 2./(Nj-1),1), (Ni,Nj,Nk))
a = T.rotate(a, (0,0,0), (0,0,1), 10.)
a = T.translate(a, (0.5,0.5,0))

ca = C.node2Center(a); ca =  C.initVars(ca, 'cellN', 1.)
cb = C.node2Center(b); cb =  C.initVars(cb, 'cellN', 1.)
res = X.optimizeOverlap(a, ca, b, cb)
C.convertArrays2File(res, "out.plt")
```

- Optimize overlapping (pyTree):

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

Ni = 50; Nj = 50; Nk = 2
a = G.cart((0,0,0),(1./(Ni-1), 1./(Nj-1),1), (Ni,Nj,Nk))
b = G.cart((0,0,0),(2./(Ni-1), 2./(Nj-1),1), (Ni,Nj,Nk)); b[0] = 'cart2'
a = T.rotate(a, (0,0,0), (0,0,1), 10.)
a = T.translate(a, (0.5,0.5,0))
t = C.newPyTree(['Base1', a, 'Base2', b])
t = X.optimizeOverlap(t)
C.convertPyTree2File(t, "out.cgns")
```

Connector.**maximizeBlankedCells**(*a*, *depth=2*, *dir=1*)

Change useless interpolated points status (2) to blanked points (0). If dir=0, uses a directional stencil of depth points, if dir=1, uses a full depth x depth x depth stencil.

Example of use:

- Maximize blanked cells (array):

```
# - maximizeBlankedCells (array) -
import Converter as C
import Connector as X
import Generator as G

def F(x,y):
    if (x+y<1): return 1
    else: return 2

Ni = 50; Nj = 50
a = G.cart((0,0,0),(1./(Ni-1),1./(Nj-1),1),(Ni,Nj,1))
a = C.initVars(a, 'cellN', F, ['x','y'])
a = X.maximizeBlankedCells(a, 2)
C.convertArrays2File([a], 'out.plt')
```

- Maximize blanked cells (pyTree):

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

```

(continues on next page)

(continued from previous page)

```
def F(x,y):
    if (x+y<1): return 1
    else: return 2

Ni = 50; Nj = 50
a = G.cart((0,0,0),(1./(Ni-1),1./(Nj-1),1),(Ni,Nj,1))
a = C.initVars(a,'cellN', F,
               ['CoordinateX','CoordinateY'])
a = C.node2Center(a, 'cellN')
a = C.rmVars(a,'cellN')
t = C.newPyTree(['Base',2]); t[2][1][2].append(a)
t = X.maximizeBlankedCells(t, 2)
C.convertPyTree2File(t, 'out.cgns')
```

Connector.**setDoublyDefinedBC**(*a*, *cellN*, *listOfInterpZones*, *listOfCelln*, *range*, *depth=2*)

When a border of zone z is defined by doubly defined BC in range=[i1,i2,j1,j2,k1,k2], one can determine whether a point is interpolated or defined by the physical BC. The array cellN defines the cell nature field at centers for zone z. If a cell is interpolable from a donor zone, then the cellN is set to 2 for this cell. The lists listOfInterpZones and listOfCelln are the list of arrays defining the interpolation domains, and corresponding cell nature fields. depth can be 1 or 2. If case of depth=2, if one point of the two layers is not interpolable, then celln is set to 1 for both points:

Example of use:

- Set interpolated/BC points on doubly defined BCs (array):

```
# - setDoublyDefinedBC (array) -
import Converter as C
import Connector as X
import Generator as G

a = G.cart((0,0,0),(1,1,1),(10,10,10))
b = G.cart((2.5,2.5,-2.5),(0.5,0.5,0.5),(10,10,30))
celln = C.array('cellN',a[2]-1,a[3]-1,a[4]-1)
celln = C.initVars(celln, 'cellN', 1)
indmax = celln[2]*celln[3]
celln[1][0][0:indmax] = 2
cellnb = C.array('cellN',b[2]-1,b[3]-1,b[4]-1)
cellnb = C.initVars(cellnb, 'cellN', 1)
celln = X.setDoublyDefinedBC(a, celln, [b], [cellnb], [1,a[2],1,a[3],1,1],␣
↪depth = 1)
ac = C.node2Center(a)
```

(continues on next page)

(continued from previous page)

```
ac = C.addVars([ac,celln])
C.convertArrays2File([ac], 'out.plt')
```

- Set interpolated/BC points on doubly defined BCs (pyTree):

```
# - setDoublyDefinedBC (pyTree) -
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G
import Transform.PyTree as T
import Converter.Internal as Internal
a = G.cart((0,0,0),(1,1,1),(10,10,10))
b = G.cart((2.5,2.5,-2.5),(0.5,0.5,0.5),(10,10,30)); b[0] = 'fente'
b = T.splitNParts(b,2)
C._addBC2Zone(a, 'overlap1', 'BCOverlap', 'kmin', zoneDonor=[
↪'FamilySpecified:FENTE'], rangeDonor='doubly_defined')
t = C.newPyTree(['Base1','Base2'])
t[2][1][2].append(a); t[2][2][2]+=b
C._addFamily2Base(t[2][2], 'FENTE')
for z in Internal.getZones(t[2][2]): C._tagWithFamily(z,'FENTE')
C._initVars(t, 'centers:cellN', 1)
t = X.applyBCOverlaps(t)
t = X.setDoublyDefinedBC(t)
C.convertPyTree2File(t, 'out.cgns')
```

Connector.**blankIntersectingCells**(*a*, *cellN*, *tol=1.e-10*)

Blank intersecting cells of a 3D mesh. Only faces normal to k-planes for structured meshes and faces normal to triangular faces for prismatic meshes, and faces normal to 1234 and 5678 faces for hexahedral meshes are tested. The cellN is set to 0 for intersecting cells/elements. Input data are A the list of meshes, cellN the list of cellNatureField located at cell centers. Array version: the cellN must be an array located at centers, defined separately

Blank intersecting cells of a 3D mesh. Only faces normal to k-planes for structured meshes and faces normal to triangular faces for prismatic meshes, and faces normal to 1234 and 5678 faces for hexahedral meshes are tested. Set the cellN to 0 for intersecting cells/elements. Input data are A the list of meshes, cellN the list of cellNatureField located at cell centers: The cellN variable is defined as a FlowSolution#Center node. The cellN is set to 0 for intersecting and negative volume cells:

a = X.blankIntersectingCells(a, tol=1.e-10, depth=2)

Example of use:

- Blank intersecting cells (array):

```
# - blankIntersectingCells (array)
import Converter as C
import Generator as G
import Transform as T
import Connector as X
a1 = G.cart((0.,0.,0.),(1.,1.,1.),(11,11,11))
a2 = T.rotate(a1, (0.,0.,0.), (0.,0.,1.),10.)
a2 = T.translate(a2, (7.,5.,5.))
A = [a1,a2]
Ac = C.node2Center(A); Ac = C.initVars(Ac,'cellN',1.);
Ac = X.blankIntersectingCells(A, Ac, tol=1.e-10)
C.convertArrays2File(Ac,"out.plt")
```

- Blank intersecting cells (pyTree):

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

a1 = G.cart((0.,0.,0.),(1.,1.,1.),(11,11,11))
a2 = T.rotate(a1, (0.,0.,0.), (0.,0.,1.),10.)
a2 = T.translate(a2, (7.,5.,5.)); a1[0] = 'cart1'; a2[0] = 'cart2'
t = C.newPyTree(['Base',a1,a2])
C._initVars(t,'centers:cellN',1.)
t2 = X.blankIntersectingCells(t, tol=1.e-10)
C.convertPyTree2File(t2, "out.cgns")
```

Connector.**getIntersectingDomains**(*t*, *t2=None*, *method='AABB'*, *taabb=None*, *tobb=None*, *taabb2=None*, *tobb2=None*)

Create a Python dictionary describing the intersecting zones. If t2 is not provided, then the computed dictionary states the self-intersecting zone names, otherwise, it computes the intersection between t and t2. Mode can be 'AABB', for Axis-Aligned Bounding Box method, 'OBB' for Oriented Bounding Box method, or 'hybrid', using a combination of AABB and OBB which gives the most accurate result. Depending on the selected mode, the user can provide the corresponding AABB and/or OBB PyTrees of t and/or t2, so that the algorithm will reuse those BB PyTrees instead of calculating them.

Example of use:

- Create intersection dictionary (array):

```
# - getIntersectingDomainsAABB (array) -
import Generator as G
import Converter as C
import Connector as X

a = G.cart((0.,0,0), (1,1,1), (10,10,10))
b = G.cart((9.,0,0), (1,1,1), (10,10,10))

bb = G.BB([a,b])
ret = X.getIntersectingDomainsAABB(bb)
print(ret)
```

- Create intersection dictionary (pyTree):

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

t = C.newPyTree(['Base1'])
Ni = 4; Nj = 4; Nk = 4; dx = 0.
for i in range(10):
    z = G.cart((dx,dx,dx),(1./(Ni-1), 1./(Nj-1),1./(Nk-1)), (Ni,Nj,Nk))
    t[2][1][2] += [z]
    dx += 0.3

interDict = X.getIntersectingDomains(t, method='hybrid')
print('Does cart.1 intersect cart.2 ?','cart.1' in interDict['cart.2'])
print('List of zones intersecting cart.2:', interDict['cart.2'])
```

Connector.**getCEBBIntersectingDomains**(*A*, *B*, *sameBase*)

Detect the domains defined in the list of bases B whose CEBB intersect domains defined in base A. Return the list of zone names for each basis. If sameBase=1, the intersecting domains are also searched in base:

Example of use:

- detect CEBB intersection between bases (pyTree):

```
# - getCEBBIntersectingDomains (pyTree) -
import Connector.PyTree as X
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal
```

(continues on next page)

(continued from previous page)

```
a = G.cart((0.,0.,0.),(0.1,0.1,0.1),(10,10,10)); a[0] = 'cart1'
b = G.cart((0.5,0.,0.),(0.1,0.1,0.1),(10,10,10)); b[0] = 'cart2'
c = G.cart((0.75,0.,0.),(0.1,0.1,0.1),(10,10,10)); c[0] = 'cart3'

t = C.newPyTree(['Cart']); t[2][1][2] += [a, b, c]
bases = Internal.getNodesFromType(t, 'CGNSBase_t')
base = bases[0]
doms = X.getCEBBIntersectingDomains(base, bases, 1); print(doms)
```

X.**getCEBBTimeIntersectingDomains**(*base*, *func*, *bases*, *funcs*, *inititer=0*, *niter=1*, *dt=1*, *sameBase*)

in a Chimera pre-processing for bodies in relative motion, it can be useful to determine intersecting domains at any iteration. niter defines the number of iterations on which CEBB intersections are detected, starting from iteration inititer. dt defines the timestep. func defines a python function defining the motion of base, funcs is the list of python functions describing motions for bases.

Warning:

1. motions here are only relative motions. If all bases are translated with the same translation motion, it must not be defined in func.
2. If no motion is defined on a basis, then the corresponding function must be []:

Example of use:

- CEBB intersection between bases with motions (pyTree):

```
# - getCEBBIntersectingDomains (pyTree) -
import Connector.PyTree as X
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal
from math import cos, sin

# Coordonnees du centre de rotation dans le repere absolu
def centerAbs(t): return [t, 0, 0]

# Coordonnees du centre de la rotation dans le repere entraine
def centerRel(t): return [5, 5, 0]

# Matrice de rotation
```

(continues on next page)

(continued from previous page)

```
def rot(t):
    omega = 30.
    m = [[cos(omega*t), -sin(omega*t), 0],
         [sin(omega*t), cos(omega*t),  0],
         [0,            0,             1]]
    return m

# Mouvement complet
def F(t): return (centerAbs(t), centerRel(t), rot(t))

a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 2., (50,50,3))
# --- CL
a = C.addBC2Zone(a,'wall','BCWall','jmin')
# --- champ aux noeuds
t = C.newPyTree(['Cylindre']); t[2][1][2].append(a)
# --- Equation state
t[2][1] = C.addState(t[2][1], 'EquationDimension', 3)
b = G.cylinder((1.5,0.,0.), 0.5, 1., 360., 0., 4., (50,50,3))
# --- champ aux centres
# --- CL
b = C.addBC2Zone(b,'wall','BCWall','jmin')
t[2][1][2].append(b); b[0]='cylinder2'
#
dt = 1.; Funcs = [F,[]]
b1 = G.cart((-2.,-2.,0.),(0.4,0.4,1),(11,11,4))
b2 = G.cart((-4.,-2.,0.),(0.4,0.4,1),(11,11,4))
b3 = G.cart((2.,-2.,0.),(0.4,0.4,1),(11,11,4))
b4 = G.cart((-2.,-2.,3.),(0.4,0.4,1),(11,11,4))
b5 = G.cart((-4.,-2.,3.),(0.4,0.4,1),(11,11,4))
b6 = G.cart((2.,-2.,3.),(0.4,0.4,1),(11,11,4))
#
t = C.addBase2PyTree(t,'Cart');t[2][2][2] = [b1,b2,b3,b4,b5,b6]
t = C.initVars(t, 'centers:cellN', 1.)
t = C.initVars(t, 'Density', 2.)
bases = Internal.getNodesFromType(t,'CGNSBase_t'); base = bases[0]
doms = X.getCEBBTimeIntersectingDomains(base, F, bases, Funcs, 0, 6, dt,␣
↪sameBase=1)
print(doms)
```

X.**applyBCOverlaps**(*t*, *depth=2*, *loc='centers'*)

set the cellN to 2 for the fringe nodes or cells (depending on parameter 'loc'='nodes' or 'centers') near the overlap borders defined in the pyTree t. Parameter 'depth' defines the number of layers of interpolated points.

Example of use:

- set cellN to 2 near overlap BCs in a pyTree (pyTree):

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

a = G.cylinder((0,0,0), 1., 1.5, 360., 0., 1., (30,30,10))
a = C.addBC2Zone(a, 'overlap1', 'BCOverlap', 'jmin')
t = C.newPyTree(['Base',a])
t = X.applyBCOverlaps(t)
C.convertPyTree2File(t, 'out.cgns')
```

X.**setDoublyDefinedBC**(*t*, *depth=2*)

when a border is defined by doubly defined BC, one can determine whether a point is interpolated or defined by the physical BC. The cellN is set to 2 if cells near the doubly defined BC are interpolable from a specified donor zone:

Example of use:

- set interpolated/BC points on doubly defined BCs (pyTree):

```
# - setDoublyDefinedBC (pyTree) -
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G
import Transform.PyTree as T
import Converter.Internal as Internal
a = G.cart((0,0,0),(1,1,1),(10,10,10))
b = G.cart((2.5,2.5,-2.5),(0.5,0.5,0.5),(10,10,30)); b[0] = 'fente'
b = T.splitNParts(b,2)
C._addBC2Zone(a, 'overlap1', 'BCOverlap', 'kmin', zoneDonor=[
↪'FamilySpecified:FENTE'], rangeDonor='doubly_defined')
t = C.newPyTree(['Base1','Base2'])
t[2][1][2].append(a); t[2][2][2]+=b
C._addFamily2Base(t[2][2], 'FENTE')
for z in Internal.getZones(t[2][2]): C._tagWithFamily(z,'FENTE')
C._initVars(t, 'centers:cellN', 1)
t = X.applyBCOverlaps(t)
t = X.setDoublyDefinedBC(t)
C.convertPyTree2File(t, 'out.cgns')
```

Connector.PyTree.**cellN2OversetHoles**(*t*)

Compute the OversetHoles node into a pyTree from the cellN field, located at nodes or centers. For structured zones, defines it as a list of ijk indices, located at nodes or centers. For unstructured zones, defines the OversetHoles node as a list of indices

ind, defining the cell vertices that are of cellN=0 if the cellN is located at nodes, and defining the cell centers that are of cellN=0 if the cellN is located at centers.

The OversetHoles nodes can be then dumped to files, defined by the indices of blanked nodes or cells.

Example of use:

- Create overset hole nodes (pyTree):

```
# - cellN2OversetHoles (pyTree) -
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G

a = G.cart((0,0,0),(1,1,1),(10,10,10))
C._initVars(a, 'centers:cellN', 0)
a = X.cellN2OversetHoles(a)
C.convertPyTree2File(a, 'out.cgns')
```

- Dump overset hole nodes to file (pyTree):

```
# - cellN2OversetHoles (pyTree) -
# - Dumping the OversetHoles node to files -
import Converter.PyTree as C
import Connector.PyTree as X
import Generator.PyTree as G
import Converter.Internal as Internal
import Converter

a = G.cart((0,0,0),(1,1,1),(10,10,10))
b = G.cart((0.5,0.5,0.5),(1,1,1),(10,10,10))
t = C.newPyTree(['Base1','Base2'])
t[2][1][2].append(a); t[2][2][2].append(b)
t = C.addBC2Zone(t, 'overlap1', 'BCOverlap', 'imin')
C._initVars(t, 'centers:cellN', 0)
t = X.applyBCOverlaps(t)
t = X.cellN2OversetHoles(t)

zones = Internal.getNodesFromType(t, 'Zone_t')
for z in zones:
    ho = Internal.getNodesFromType(z, 'OversetHoles_t')
    if ho != []:
        h = ho[0][2][1][1]
        array = ['cell_index', h, h.size, 1, 1]
        Converter.convertArrays2File([array], 'hole_'+z[0]+'.v3d',
                                     'bin_v3d')
```

Connector.PyTree.**setInterpData**(*aR*, *aD*, *double_wall=0*, *order=2*, *penalty=1*, *nature=0*, *loc='nodes'*, *storage='direct'*, *topTreeRcv=None*, *topTreeDnr=None*, *sameName=0*)

Compute and store in a pyTree the interpolation information (donor and receptor points, interpolation type, interpolation coefficients) given receptors defined by aR, donor zones given by aD. If storage='direct', then aR with interpolation data stored in receptor zones are returned, and if storage='inverse', then aD with interpolation data stored in donor zones are returned. Donor zones can be structured or unstructured TETRA. receptor zones can be structured or unstructured.

Interpolation order can be 2, 3 or 5 for structured donor zones, only order=2 for unstructured donor zones is performed.

Parameter loc can 'nodes' or 'centers', meaning that receptor points are zone nodes or centers.

penalty=1 means that a candidate donor cell located at a zone border is penalized against interior candidate cell.

nature=0 means that a candidate donor cell containing a blanked point(cellN=0) is not valid. If nature=1 all the nodes of the candidate donor cell must be cellN=1 to be valid.

double_wall=1 activates the double wall correction. If there are walls defined by families in aR or aD, the corresponding top trees topTreeRcv or/and topTreeDnr must be defined.

If sameName=1, interpolation from donor zones with the same name as receptor zones are avoided.

> **Warning:** currently, no periodic Chimera taken into account by this function automatically.

Interpolation data are stored as a ZoneSubRegion_t node, stored under the donor or receptor zone node depending of the storage.

Example of use:

- Compute interpolation connectivity (pyTree):

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

(continues on next page)

(continued from previous page)

```
a = G.cylinder((0,0,0),1.,3.,360,0,1,(200,30,2)); a[0] = 'cylindre1'
a = C.addBC2Zone(a, 'wall1', 'BCWall', 'jmin')
a = C.addBC2Zone(a, 'ov1', 'BCOverlap', 'jmax')
b = G.cylinder((4,0,0),1.,3.,360,0,1,(200,30,2)); b[0] = 'cylindre2'
b = C.addBC2Zone(b, 'wall1', 'BCWall', 'jmin')
b = C.addBC2Zone(b, 'ov1', 'BCOverlap', 'jmax')
c = G.cart((-5.,-7.5,0), (15./200,15./200,1), (200,200,2))
t = C.newPyTree(['Corps1', 'Corps2', 'Bgrd'])
t[2][1][2].append(a); t[2][2][2].append(b); t[2][3][2].append(c)
t = X.connectMatch(t, dim=2)
t = C.fillEmptyBCWith(t, 'nref', 'BCFarfield', dim=2)
t = X.applyBCOverlaps(t, depth=1)
t[2][2:] = X.setInterpData(t[2][1], t[2][2:], loc='centers', storage='inverse')
C.convertPyTree2File(t, "out.cgns")
```

Connector.PyTree.**getOversetInfo**(*tR*, *tD*, *type='interpolated'*)

Set information on Chimera connectivity, i.e. interpolated, extrapolated or orphan cells, donor aspect ratio and ratio between volume of donor and receptor cells. This function is compliant with the storage as defined for setInterpData function. If type='interpolated', variable 'interpolated' is created and is equal to 1 for interpolated and extrapolated points, 0 otherwise. If type='extrapolated', variable 'extrapolated' is created and its value is the sum of the absolute values of coefficients, 0 otherwise. If type='orphan', variable 'orphan' is created and is equal to 1 for orphan points, 0 otherwise. If type='cellRatio', variable 'cellRatio' is created and is equal to max(volD/volR,volR/volD) for interpolated and extrapolated points (volR and volD are volume of receptors and donors). If type='donorAspect', variable 'donorAspect' is created and is equal to the ratio between the maximum and minimum length of donors, and 0 for points that are not interpolated.

Example of use:

- Get overset information (pyTree):

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

a = G.cylinder((0,0,0),1.,3.,360,0,1,(200,30,4)); a[0] = 'cylindre1'
a = C.addBC2Zone(a, 'wall1', 'BCWall', 'jmin')
a = C.addBC2Zone(a, 'ov1', 'BCOverlap', 'jmax')
b = G.cylinder((4,0,0),1.,3.,360,0,1,(200,30,4)); b[0] = 'cylindre2'
b = C.addBC2Zone(b, 'wall1', 'BCWall', 'jmin')
```

(continues on next page)

(continued from previous page)

```
b = C.addBC2Zone(b, 'ov1', 'BCOverlap', 'jmax')
c = G.cart((-5.,-7.5,-2), (15./200,15./200,1), (200,200,8))
t = C.newPyTree(['Corps1', 'Corps2', 'Bgrd'])
t[2][1][2].append(a); t[2][2][2].append(b); t[2][3][2].append(c)
t = X.connectMatch(t, dim=3)
t = C.fillEmptyBCWith(t, 'nref', 'BCFarfield', dim=3)
t = X.applyBCOverlaps(t, depth=1,loc='centers')
t[2][1][2] = X.setInterpData(t[2][1][2], t, loc='centers',
                             storage='direct', sameName=1)
t = X.getOversetInfo(t, t, loc='centers', type='interpolated')
t = X.getOversetInfo(t, t, loc='centers', type='orphan')
C.convertPyTree2File(t, "out.cgns")
```

Connector.PyTree.**extractChimeraInfo**(*t*, *type='interpolated'*, *loc='centers'*)

Extract some Chimera information (interpolated, extrapolated, orphan points or extrapolated points with a sum of coefficients greater than a given value). Function chimeraInfo or oversetInfo must be applied first to compute the interpolated/extrapolated/orphan fields. Information is extracted as 'NODE'-type zones, whose names are suffixed by the original zone names. If no zone is extracted, returns []. Location loc must be compliant with the location of interpolation data (i.e. location of Chimera receptor points).

If type='interpolated', interpolated and extrapolated points are extracted. If type='extrapolated', extrapolated points are extracted. If type='orphan', orphan points are extracted. If type='cf>value', extrapolated points where the sum of absolute coefficients is greater than value are extracted.

Example of use:

- Extract chimera points (pyTree):

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

a = G.cylinder((0,0,0),1.,3.,360,0,1,(200,30,4)); a[0] = 'cylindre1'
C._addBC2Zone(a, 'wall1', 'BCWall', 'jmin')
C._addBC2Zone(a, 'ov1', 'BCOverlap', 'jmax')
b = G.cylinder((4,0,0),1.,3.,360,0,1,(200,30,4)); b[0] = 'cylindre2'
C._addBC2Zone(b, 'wall1', 'BCWall', 'jmin')
C._addBC2Zone(b, 'ov1', 'BCOverlap', 'jmax')
t = C.newPyTree(['Cyl1',a,'Cyl2',b])
t = X.connectMatch(t, dim=3)
C._fillEmptyBCWith(t,'nref','BCFarfield', dim=3)
```

(continues on next page)

(continued from previous page)

```
C._initVars(t[2][2],'centers:cellN',2)
t = X.setInterpolations(t, loc='cell', double_wall=1, storage='direct')
X._chimeraInfo(t, type='orphan')
orphanPts = X.extractChimeraInfo(t,type='orphan')
C.convertPyTree2File(orphanPts, "orphanPts.cgns")
X._chimeraInfo(t,type='extrapolated')
out = X.extractChimeraInfo(t,type='cf>1.5')
C.convertPyTree2File(out,"out.cgns")
```

3.3 Overset grid connectivity for elsA solver

Connector.PyTree.**setInterpolations**(*t, loc='cell', double_wall=0, storage='inverse', prefixFile='', sameBase=0, solver='elsA', nGhostCells=2, parallelDatas=[], cfMax=30., planarTol=0., check=True*)

This function is specific to elsA solver. Set the Chimera connectivity (EX points and cell centers to be interpolated, index for donor interpolation cell and interpolation coefficients). Double wall projection technique is activated if 'double_wall=1'. Parameter planarTol can be useful for double wall cases, in the case when double wall surfaces are planar but distant from planarTol to each other. Parameter 'sameBase=1' means that donor zones can be found in the same base as the current zone.

parallelDatas=[graph,rank,listOfInterpCells] is defined only in a coupling context. It contains the graph of communication, the rank of the current processor and the list of interpolated cells/faces indices. graph is a Python dictionary with the following structure : graph[proc1][proc2] gives the list of zones on processeur 1 which intersect with zones on processor 2. loc='cell' or 'face' indicates the location of the interpolated points (cell center or face). Interpolations with location at 'face' correspond to interpolation for EX points according to elsA solver.

storage can be 'direct' (interpolation data stored in receptor zones) or 'inverse' (data stored in donor zones). Value storage='direct' is only valid if Chimera connectivity is read by elsA solver as Chimera connectivity files.

In a distributed mode, the storage must be necessarily 'inverse'. If the Chimera connectivity is read by elsA directly as CGNS ZoneSubRegion_t nodes, then the storage must be 'inverse' too. prefixFile is the prefix for the name of the connectivity files generated for elsA solver (solver='elsA') or Cassiopee solver (solver='Cassiopee'). If prefixFile is not specified by the user, no Chimera connectivity file is dumped.

nGhostCells is the number of ghost cells that are required by elsA solver when writing connectivity files. Can be greater than 2 only if elsA reads the Chimera connectivity

files. cfMax is a threshold value for a valid extrapolation: if the sum of the extrapolated coefficients is greater than cfMax, then the cell is marked as orphan.

parallelDatas: used to perform setInterpolations in a distributed context : a list of communication information [graph, rank, interpPts], where interpPts is the list of interpolated cells/EX points.

Parameter check is a Boolean which displays the summary of interpolated, extrapolated and orphan points.

Exists also as an in-place function (X._setInterpolations):

Example of use:

- set interpolations (pyTree):

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

a = G.cylinder((0,0,0),1.,3.,360,0,1,(200,30,2)); a[0] = 'cylindre1'
a = C.addBC2Zone(a, 'wall1', 'BCWall', 'jmin')
a = C.addBC2Zone(a, 'ov1', 'BCOverlap', 'jmax')
b = G.cylinder((4,0,0),1.,3.,360,0,1,(200,30,2)); b[0] = 'cylindre2'
b = C.addBC2Zone(b, 'wall1', 'BCWall', 'jmin')
b = C.addBC2Zone(b, 'ov1', 'BCOverlap', 'jmax')
c = G.cart((-5.,-7.5,0), (15./200,15./200,1), (200,200,2))
t = C.newPyTree(['Corps1', 'Corps2', 'Bgrd'])
t[2][1][2].append(a); t[2][2][2].append(b);t[2][3][2].append(c)
t = X.connectMatch(t, dim=2)
t = C.fillEmptyBCWith(t,'nref','BCFarfield', dim=2)
t = X.applyBCOverlaps(t, depth=1)
t = X.setInterpolations(t, loc='cell', prefixFile='chimData')
C.convertPyTree2File(t, "out.cgns")
```

Connector.PyTree.**chimeraTransfer**(*t, storage='direct', variables=[], loc='cell', mesh='extended'*)

Compute Chimera transfers. This function is compliant with the storage as it is defined for setInterpolations function.

Parameter storage can be 'direct' or 'inverse' and must be consistent with the storage computed by setInterpolations

Parameter 'variables' specifies the variables for which the transfer is applied.

Parameter loc can be 'cell' or 'face' to transfer the variables at cell centers or EX points.

Parameter mesh can be 'standard' or 'extended'. For elsA simulations, it is mandatory to use mesh='extended' and storage='inverse'.

Exists also as an in-place function (X._chimeraTransfer)

Example of use:

- compute Chimera transfers (pyTree):

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

a = G.cylinder((0,0,0),1.,3.,360,0,1,(200,30,3)); a[0] = 'cylindre1'
C._addBC2Zone(a, 'wall1', 'BCWall', 'jmin')
C._addBC2Zone(a, 'ov1', 'BCOverlap', 'jmax')
b = G.cylinder((4,0,0),1.,3.,360,0,1,(200,30,3)); b[0] = 'cylindre2'
C._addBC2Zone(b, 'wall1', 'BCWall', 'jmin')
C._addBC2Zone(b, 'ov1', 'BCOverlap', 'jmax')
c = G.cart((-5.,-7.5,0), (15./200,15./200,1), (200,200,3))
t = C.newPyTree(['Corps1','Corps2','Bgrd'])
t[2][1][2].append(a); t[2][2][2].append(b); t[2][3][2].append(c)
t = X.connectMatch(t, dim=2)
C._fillEmptyBCWith(t,'nref','BCFarfield', dim=3)
X._applyBCOverlaps(t, depth=2)
t = X.setInterpolations(t, storage = 'direct')
for i in range(len(t[2])):
    C._initVars(t[2][i], 'centers:Density', float(i+1))
    C._initVars(t[2][i], 'centers:MomentumX', float(i+1))
    C._initVars(t[2][i], 'centers:MomentumY', float(i+1))
    C._initVars(t[2][i], 'centers:MomentumZ', float(i+1))
    C._initVars(t[2][i], 'centers:StagnationEnergy', float(i+1))
t = X.chimeraTransfer(t, storage='direct', variables=['centers:Density',
↪'centers:MomentumX','centers:MomentumY','centers:MomentumZ',
↪'centers:StagnationEnergy'])
C.convertPyTree2File(t, 'out.cgns')
```

Connector.PyTree.**chimeraInfo**(*t*, *type='interpolated'*)

Set information on Chimera connectivity, i.e. interpolated, extrapolated or orphan cells, donor aspect ratio and ratio between volume of donor and receptor cells.

This function is compliant with the storage as it is defined for setInterpolations function.

If type='interpolated', variable 'centers:interpolated' is created and is equal to 1 for interpolated and extrapolated cells, 0 otherwise.

If type='extrapolated', variable 'centers:extrapolated' is created and its value is the sum of the absolute values of coefficients, 0 otherwise.

If type='orphan', variable 'centers:orphan' is created and is equal to 1 for orphan cells, 0 otherwise.

If type='cellRatio', variable 'centers:cellRatio' is created and is equal to max(volD/volR,volR/volD) for interpolated and extrapolated cells (volR and volD are volume of receptor and donor cells).

If type='donorAspect', variable 'centers:donorAspect' is created and is equal to the ratio between the maximum and minimum length of donor cells, and 0 for cells that are not interpolated.

Exists also as an in-place function (X._chimeraInfo)

Example of use:

- set Chimera information (pyTree):

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

a = G.cylinder((0,0,0),1.,3.,360,0,1,(200,30,4)); a[0] = 'cylindre1'
a = C.addBC2Zone(a, 'wall1', 'BCWall', 'jmin')
a = C.addBC2Zone(a, 'ov1', 'BCOverlap', 'jmax')
b = G.cylinder((4,0,0),1.,3.,360,0,1,(200,30,4)); b[0] = 'cylindre2'
b = C.addBC2Zone(b, 'wall1', 'BCWall', 'jmin')
b = C.addBC2Zone(b, 'ov1', 'BCOverlap', 'jmax')
c = G.cart((-5.,-7.5,-2), (15./200,15./200,1), (200,200,8))
t = C.newPyTree(['Corps1', 'Corps2', 'Bgrd'])
t[2][1][2].append(a); t[2][2][2].append(b); t[2][3][2].append(c)
t = X.connectMatch(t, dim=3)
t = C.fillEmptyBCWith(t,'nref','BCFarfield', dim=3)
t = X.applyBCOverlaps(t, depth=1)
t = X.setInterpolations(t, loc='cell', double_wall=1, storage='direct')
t = X.chimeraInfo(t, type='interpolated')
C.convertPyTree2File(t, "out.cgns")
```

3.4 Immersed boundary (IBM) pre-processing

Connector.PyTree.**setIBCData**(*aR*, *aD*, *order=2*, *penalty=0*, *nature=0*, *method='lagrangian'*, *loc='nodes'*, *storage='direct'*, *he=0.*, *hi=0.*, *dim=3*)

Compute and store IBM information (donor and receptor points, interpolation type, interpolation coefficients, coordinates of corrected, wall and interpolated points) given receptors defined by aR, donor zones given by aD.

- If storage='direct', then aR with interpolation data stored in receptor zones are returned, and if storage='inverse', then aD with interpolation data stored in donor zones are returned.
- Donor zones can be structured or unstructured TETRA. receptor zones can be structured or unstructured ;
- Interpolation order can be 2, 3 or 5 for structured donor zones, only order=2 for unstructured donor zones is performed ;
- Parameter loc can 'nodes' or 'centers', meaning that receptor points are zone nodes or centers ;
- penalty=1 means that a candidate donor cell located at a zone border is penalized against interior candidate cell ;
- nature=0 means that a candidate donor cell containing a blanked point(cellN=0) is not valid. If nature=1 all the nodes of the candidate donor cell must be cellN=1 to be valid ;
- Interpolation data are stored as a ZoneSubRegion_t node, stored under the donor or receptor zone node depending of the storage ;
- aR must contain information about distances and normals to bodies, defined by 'TurbulentDistance','gradxTurbulentDistance','gradyTurbulentDistance' and 'gradzTurbulentDistance', located at nodes or cell centers ;
- Corrected points are defined in aR as points with cellN=2, located at nodes or cell centers ;
- Parameter he is a constant, meaning that the interpolated points are pushed away of a distance he from the IBC points if these are external to the bodies ;
- Parameter hi is a constant. If hi=0., then the interpolated points are mirror points of IBC points. If hi>0., then these mirror points are then pushed away of hi from their initial position ;
- hi and he can be defined as a field (located at nodes or centers) for any point.

Example of use:

- set IBM data in the pyTree (pyTree):

```
# - setIBCData (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X
import Post.PyTree as P
import numpy as N
import Dist2Walls.PyTree as DTW
import Transform.PyTree as T

a = G.cart((-1,-1,-1),(0.01,0.01,1),(201,201,3))
s = G.cylinder((0,0,-1), 0, 0.4, 360, 0, 4, (30,30,5))
s = C.convertArray2Tetra(s); s = T.join(s); s = P.exteriorFaces(s)
t = C.newPyTree(['Base',a])
# Blanking
bodies = [[s]]
BM = N.array([[1]],N.int32)
t = X.blankCells(t,bodies,BM,blankingType='center_in')
t = X.setHoleInterpolatedPoints(t, depth=-2)
# Dist2Walls
DTW._distance2Walls(t,[s],type='ortho',loc='centers',signed=1)
t = C.center2Node(t,'centers:TurbulentDistance')
# Gradient de distance localise en centres => normales
t = P.computeGrad(t, 'TurbulentDistance')
t = X.setIBCData(t, t, loc='centers', storage='direct',hi=0.03)
C.convertPyTree2File(t, "out.cgns")
```

Connector.ToolboxIBM.**prepareIBMData**(*t*, *tb*, *DEPTH=2*, *loc='centers'*, *frontType=1*)

Compute and store all the information required for IBM computations. For Euler computations, corrected points are inside body, for viscous computations, corrected points are outside body.

Parameters

- **t** (pyTree) – pyTree defining the computational domain as a structured mesh
- **tb** (pyTree) – pyTree defining the obstacles. Each obstacle must be defined in a CGNS basis as a surface closed mesh, whose normals must be oriented towards the fluid region
- **DEPTH** (integer) – number of layers of IBM corrected points (usually 2 meaning 2 ghost cells are required)
- **loc** (string) – location of IBM points: at nodes if loc='nodes' or at cell centers if loc='centers' ('centers'default)

- **frontType** (integer [0-2]) – 0: constant distance front, 1: minimum distance front, 2: adapted distance front

Returns t, tc

Return type pyTree

The problem dimension (2 or 3D) and the equation model (Euler, Navier-Stokes or RANS) are required in tb.

Output datas are:

- the pyTree t with a 'cellN' field located at nodes or centers, marking as computed/updated/blanked points for the solver (cellN=1/2/0).
- a donor pyTree tc storing all the information required to transfer then the solution at cellN=2 points:

Example of use:

- compute the IBM preprocessing (pyTree):

```
# - prepareIBMData (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.ToolboxIBM as IBM
import Post.PyTree as P
import Geom.PyTree as D
import Dist2Walls.PyTree as DTW
N = 51
a = G.cart((0,0,0),(1./(N-1),1./(N-1),1./(N-1)),(N,N,N))
body = D.sphere((0.5,0,0),0.1,N=20)
t = C.newPyTree(['Base', a])
tb = C.newPyTree(['Base', body])
tb = C.addState(tb, 'EquationDimension',3)
tb = C.addState(tb, 'GoverningEquations', 'NSTurbulent')
DTW._distance2Walls(t,bodies=tb,loc='centers',type='ortho')
t = P.computeGrad(t,'centers:TurbulentDistance')
t,tc=IBM.prepareIBMData(t,tb, DEPTH=2,frontType=1)
C.convertPyTree2File(t,'t.cgns')
C.convertPyTree2File(t,'tc.cgns')
```

Warning: requires Connector.ToolboxIBM.py module.

Connector.ToolboxIBM.**extractIBMInfo**(*a*)

Extract the IBM particular points once the IBM data is computed and stored in a

pyTree. These points are the IBM points that are marked as updated points for the IBM approach and the corresponding wall and interpolated (in fluid) points.

If information is stored in the donor pyTree tc, then a=tc, else a must define the receptor pyTree t.

Example of use:

- extract all the IBM points (pyTree):

```
# - prepareIBMData (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.ToolboxIBM as IBM
import Post.PyTree as P
import Geom.PyTree as D
import Dist2Walls.PyTree as DTW

N = 21
a = G.cart((0,0,0),(1./(N-1),1./(N-1),1./(N-1)),(N,N,N))
body = D.sphere((0.5,0,0),0.1,N=20)
t = C.newPyTree(['Base',a])
tb = C.newPyTree(['Base',body])
tb = C.addState(tb, 'EquationDimension',3)
tb = C.addState(tb, 'GoverningEquations', 'NSTurbulent')
t = DTW.distance2Walls(t,bodies=tb,loc='centers',type='ortho')
t = P.computeGrad(t,'centers:TurbulentDistance')
t,tc=IBM.prepareIBMData(t,tb, DEPTH=2)
res = IBM.extractIBMInfo(tc)
C.convertPyTree2File(res,"res.cgns")
```

Warning: requires Connector.ToolboxIBM.py module.

Connector.ToolboxIBM.**extractIBMWallFields**(*a*, *tb=None*)

Extract the solution at walls. If IBM data is stored in donor pyTree tc, then a must be tc, else a is the pyTree t.

If tb is None, then tw is the cloud of wall points. If tb is a triangular surface mesh, then the solution extracted at cloud points is interpolated on the vertices of the triangular mesh. a must contain the fields in the ZoneSubRegions of name prefixed by 'IBCD'.

Example of use:

- Extract the solution on the wall (pyTree):

```
# - extractionIBM a la paroi (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X
import Geom.PyTree as D
import Post.PyTree as P
import Dist2Walls.PyTree as DTW
import Transform.PyTree as T
import Initiator.PyTree as I
import Converter.Internal as Internal
import Connector.ToolboxIBM as IBM
import KCore.test as test
import numpy

N = 41
a = G.cart((0,0,0),(1./(N-1),1./(N-1),1./(N-1)),(N,N,N))
xm = 0.5*N/(N-1)
s = D.sphere((xm,xm,xm),0.1,N=20)
s = C.convertArray2Tetra(s); s = G.close(s)
t = C.newPyTree(['Base', a])

# Blanking
bodies = [[s]]
BM = numpy.array([[1]],numpy.int32)
t = X.blankCells(t,bodies,BM,blankingType='center_in')
X._setHoleInterpolatedPoints(t,depth=-2)
# Dist2Walls
DTW._distance2Walls(t,[s],type='ortho',loc='centers',signed=1)
t = C.center2Node(t,'centers:TurbulentDistance')
# Gradient de distance localise en centres => normales
t = P.computeGrad(t, 'TurbulentDistance')
C._initVars(t,"centers:Density",1.)
C._initVars(t,"centers:VelocityX",0.2)
C._initVars(t,"centers:VelocityY",0.)
C._initVars(t,"centers:VelocityZ",0.)
C._initVars(t,"centers:Temperature",1.)
tc = C.node2Center(t)

tb = C.newPyTree(['Base', s])
C._addState(tb, 'EquationDimension',3)
C._addState(tb, 'GoverningEquations', 'NSTurbulent')

X._setIBCData(t, tc, loc='centers', storage='inverse', bcType=0)

#test avec arbre tc compact
vars=['Density','VelocityX','VelocityY','VelocityZ','Temperature']
```

(continues on next page)

(continued from previous page)

```
zones = Internal.getNodesFromType2(t, 'Zone_t')
X.miseAPlatDonorTree__(zones, tc, graph=None)
# attention compact=0 car t n est pas compacte
X._setInterpTransfers(t,tc, bcType=0,varType=2,variablesIBC=vars,compact=0,
↪compactD=1)
z = IBM.extractIBMWallFields(tc, tb=tb)
C.convertPyTree2File(z,"out.cgns")
```

Warning: requires Connector.ToolboxIBM.py module.

CHAPTER

FOUR

OVERSET AND IMMERSED BOUNDARY TRANSFERS WITH PYTREES

The following function enables to update the solution at some points, marked as interpolated for overset and IBM approaches.

Connector.PyTree.**setInterpTransfers**(*aR, topTreeD, variables=None, variablesIBC=['Density', 'MomentumX', 'MomentumY', 'MomentumZ', 'EnergyStagnationDensity'], bcType=0, varType=1, storage='unknown'*)

General transfers from a set of donor zones defined by topTreeD to receptor zones defined in aR.

Both Chimera and IBM transfers can be applied and are identified by the prefix name of the ZoneSubRegion node created when computing the overset or IBM interpolation data.

Parameter variables is the list of variable names that are transfered by Chimera interpolation.

Parameter variablesIBC defines the name of the 5 variables used for IBM transfers.

Parameter bcType can be 0 or 1 (see table below for details).

Parameter varType enables to define the meaning of variablesIBC, if their name is not standard (see table below for more details).

Parameter storage enables to define how the information is stored (see table below).

Parameter value	Meaning
bcType=0	IBM transfers model slip conditions
bcType=1	IBM transfers model no-slip conditions
varType=1	Density,MomentumX,MomentumY,MomentumZ,EnergyStagnationDensity
varType=2	Density,VelocityX,VelocityY,VelocityZ,Temperature
varType=3	Density,VelocityX,VelocityY,VelocityZ,Pressure
storage=0	Interpolation data is stored in receptor zones aR
storage=1	Interpolation data is stored in donor zones topTreeD
storage=-1	Interpolation data storage is unknown or can be stored in donor and receptor zones.
extract=1	Wall fields are stored in zone subregions (density and pressure, utau and yplus if wall law is applied).

Exists also as an in-place function (X._setInterpTransfers):

Example of use:

- Transfers the solution from donor zones to receptor zones (pyTree):

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

a = G.cylinder( (0,0,0), 1, 2, 0, 360, 1, (60, 20, 3) )
b = G.cylinder( (0,0,0), 1, 2, 3, 160, 1, (30, 10, 3) )
a = C.addBC2Zone(a, 'wall', 'BCWall', 'jmin')
a = C.addBC2Zone(a, 'match', 'BCMatch', 'imin', a, 'imax', trirac=[1,
↪2,3])
a = C.addBC2Zone(a, 'match', 'BCMatch', 'imax', a, 'imin', trirac=[1,
↪2,3])
b = C.addBC2Zone(b, 'wall', 'BCWall', 'jmin')
b = C.addBC2Zone(b, 'overlap', 'BCOverlap', 'imin')
b = C.addBC2Zone(b, 'overlap', 'BCOverlap', 'imax')
t1 = C.newPyTree(['Base']); t1[2][1][2] = [a];
t2 = C.newPyTree(['Base']); t2[2][1][2] = [b]
```

(continues on next page)

(continued from previous page)

```
t1 = C.fillEmptyBCWith(t1, 'nref', 'BCFarfield')
t2 = C.fillEmptyBCWith(t2, 'nref', 'BCFarfield')

t1 = C.initVars(t1, '{Density}= 1.')
t2 = C.initVars(t2, '{Density}=-1.')
t2 = C.node2Center(t2, ['Density'])

t2 = X.applyBCOverlaps(t2, depth=1)
t1 = X.setInterpData(t2, t1, double_wall=1, loc='centers',
                     storage='inverse', order=3)
t2 = X.setInterpTransfers(t2, t1, variables=['Density'])
C.convertPyTree2File(t2, 'out.cgns')
```

4.1 Index

- genindex
- modindex
- search