ONERA

THE FRENCH AEROSPACE LAB

Post Documentation

Release 3.7

/ELSA/MU-10019/V3.7

Oct 19, 2023

CONTENTS

CHAPTER

ONE

PREAMBLE

This module provides post-processing tools for CFD simulations. It manipulates arrays (as defined in Converter documentation) or CGNS/Python trees (or pyTree, as defined in Converter/Internal documentation) as data structures.

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

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

```
import Post
```

For use with the pyTree interface:

```
import Post.PyTree as Post
```

CHAPTER

TWO

LIST OF FUNCTIONS

– Modifying/creating Variables

Post.renameVars(array, varsPrev, varsNew)	Rename variables names in varsPrev with names defined by varsNew.
Post.PyTree.importVariables(t1, t2[, ...])	Import the variables of tree t1 to tree t2.
Post.computeVariables(array, varname[, ...])	Compute the variables defined in varname for array.
Post.computeExtraVariable(array, varname[, ...])	Compute variables that require a change of location.
Post.PyTree.computeWallShearStress(t)	Compute wall shear stress.
Post.computeGrad(array, varname)	Compute the gradient of the field varname defined in array.
Post.computeGrad2(array, arrayc[, vol, ...])	Compute the gradient of a field defined on centers.
Post.computeGradLSQ(array, arrayc, dim)	Compute gradient by least mean square.
Post.computeHessian(array, arrayc, dim)	Compute hessian of array.
Post.computeNormGrad(array, varname)	Compute the norm of gradient of field varname defined in array.
Post.computeDiv(array, vector)	Compute the divergence of the given vector, whose components are defined in array using the computeGrad method for gradients.
Post.computeDiv2(array, arrayc[, vol, ...])	Compute the divergence of the field varname, whose components are defined in array using the computeGrad2 method for gradients.
Post.computeCurl(array, vector)	Compute the curl of the 3D-field defined in array.
Post.computeNormCurl(array, vector)	Compute the norm of the curl of the 3D-field defined in array.
Post.computeDiff(array, varname)	Compute the difference of the field varname defined in array.

– Solution selection

Post.selectCells(arrayNodes, F[, ...])	Select cells in a given array.
Post.selectCells2(an, tag[, ac, strict, ...])	Select cells in a given array following tag.
Post.interiorFaces(a[, strict])	Interior faces of an array a.
Post.exteriorFaces(a[, indices])	Exterior faces of an array.
Post.exteriorFacesStructured(a)	Return the list of exterior faces for a structured array Usage: exteriorFacesStructured(a,indices)
Post.exteriorElts(array)	Exterior elements of an array.
Post.frontFaces(a, tag)	Select faces located in the front of tag=0 and tag=1.
Post.sharpEdges(array[, alphaRef])	Detect sharp edges between adjacent cells of a surface.
Post.silhouette(array, vector)	Detect shape of an unstructured surface.
Post.coarsen(a, indic[, argqual, tol])	Coarsen a surface TRI-type mesh given a coarsening indicator for each element.
Post.refine(a[, indic, w])	Refine a surface TRI-type mesh given an indicator for each element.
Post.computeIndicatorValue(octreeHexa, ...)	Computes the indicator value on the octree mesh based on the maximum value of the absolute value of indicField.
Post.computeIndicatorField(octreeHexa, indicVal)	Compute the indicator -1, 0 or 1 for each element of the HEXA octree with respect to the indicatorValue field located at element centers.

– Solution extraction

Post.extractPoint(arrays, Pts[, order, ...])	Extract the solution in one or more points.
Post.extractPlane(arrays, T[, order, tol])	Slice solution with a plane.
Post.extractMesh(arrays, extractArray[, ...])	Extract the solution on a given mesh.
Post.projectCloudSolution(cloudArray, surfArray)	Project the solution defined on a set of points to a TRI surface.
Post.zipper(arrays[, options])	Extract Chimera surface as an unique unstructured surface.
Post.usurp(blkArrays[, ibArrays])	Extract unique surfaces using ranked polygons.
Post.Probe.Probe(fileName[, t, X, ind, ...])	Probe for extracting data from fields during computations.
Post.Probe.Probe.extract(t, time[, ...])	Extract XYZ or Ind fields from t.
Post.Probe.Probe.flush()	Flush probe to file.
Post.Probe.Probe.read([cont, ind, probeName])	Reread all data from probe file.

– Streams/Isos

Post.streamLine(arrays, X0, vector[, N, dir])	Compute a streamline starting from (x0,y0,z0) given a list of arrays containing 'vector' information.
Post.streamRibbon(arrays, X0, N0, vector[, ...])	Compute a streamribbon starting from (x0,y0,z0) given a list of arrays containing 'vector' information.
Post.streamSurf(arrays, b, vector[, N, dir])	Compute a stream surface.
Post.isoLine(array, var, value)	Compute an isoLine correponding to value of field 'var' on arrays.
Post.isoSurf(array, var, value[, split])	Compute an isoSurf corresponding to value of field 'var' in volume arrays.
Post.isoSurfMC(array, var, value[, split])	Compute an isoSurf correponding to value of field 'var' in volume arrays.

– Solution integration

Post.integ(coordArrays, FArrays, ratioArrays)	Integral of fields.
Post.integNorm(coordArrays, FArrays, ratioArrays)	Integral of fields times normal.
Post.integNormProduct(coordArrays, FArrays, ...)	Integral of scalar product fields times normal.
Post.integMoment(coordArrays, FArrays, ...)	Integral of moments.
Post.integMomentNorm(coordArrays, FArrays, ...)	Integral of moments (OM^f.vect(n)).

CHAPTER

THREE

CONTENTS

3.1 Modifying/creating variables

Post.**renameVars**(*t*, *oldVarNameList*, *newVarNameList*)

Rename a list of variables with new variable names. Exists also as in place function (_renameVars) that modifies t and returns None.

Parameters

- **t** (*[array, arrays] or [zone, list of zones, base, tree]*) – Input data
- **oldVarNameList** (*list of strings*) – list of variables to rename
- **newVarNameList** (*list of strings*) – list of new variable names

Returns

reference copy of input

Return type

identical to t

Example of use:

- Rename variables (array):

```
# - renameVars (array) -
import Converter as C
import Post as P
import Generator as G

ni = 30; nj = 40
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,2))
m = C.initVars(m, 'ro', 1.)
m = C.initVars(m, 'rou', 1.)
```

(continues on next page)

(continued from previous page)

```
# Rename a list of variables
m2 = P.renameVars(m, ['ro','rou'], ['Density','MomentumX'])
C.convertArrays2File(m2, "out.plt")
```

- Rename variables (pyTree):

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

ni = 30; nj = 40
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,2))
m = C.addVars(m, ['Density', 'centers:MomentumX'])

# Rename a list of variables
m2 = P.renameVars(m, ['Density', 'centers:MomentumX'], ['Density_M',
↪'centers:MomentumX_M'])

C.convertPyTree2File(m2, 'out.cgns')
```

Post.PyTree.**importVariables**(*t1*, *t2*, *method=0*, *eps=1.e-6*, *addExtra=1*)

Variables located at nodes and/or centers can be imported from a pyTree t1 to a pyTree t2. If one variable already exists in t2, it is replaced by the same variable from t1. If method=0, zone are matched from names, if method=1, zones are matched from coordinates with a tolerance eps, if method=2, zones are taken in the given order of t1 and t2 (must match one by one). If addExtra=1, unmatched zones are added to a base named 'EXTRA'.

Parameters

- **t1** (*pyTree*) – Input data
- **t2** (*pyTree*) – Input data

Returns

reference copy of t2

Return type

pyTree

Example of use:

- Import variables to a tree (pyTree):

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

z1 = G.cart((0.,0.,0.),(0.1,0.1,0.1),(10,10,10))
z2 = G.cart((0.,0.,0.),(0.1,0.1,0.1),(10,10,10))
t1 = C.newPyTree(['Base', z1]); t2 = C.newPyTree(['Base', z2])

C._initVars(t1,'centers:cellN',1.)
C._initVars(t1,'centers:Density',1.)
C._initVars(t1,'Pressure',10.)
C._initVars(t2,'centers:cellN',0.)

t2 = P.importVariables(t1, t2)

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

Post.**computeVariables**(*a*, *varList*, *gamma=1.4*, *rgp=287.053*, *s0=0.*, *betas=1.458e-6*, *Cs=110.4*, *mus=1.76e-5*, *Ts=273.15*)

New variables can be computed from conservative variables. The list of the names of the variables to compute must be provided. The computation of some variables (e.g. viscosity) require some constants as input data. In the pyTree version, if a reference state node is defined in the pyTree, then the corresponding reference constants are used. Otherwise, they must be specified as an argument of the function. Exists also as in place version (_computeVariables) that modifies a and returns None.

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – Input data
- **varList** (*list of strings*) – list of variable names (can be preceded by 'nodes:' or 'centers:')

Return type

identical to input

The constants are:

- 'gamma' for the specific heat ratio: γ;
- 'rgp' for the perfect gas constant: $R = (\gamma - 1) \times C_v$;
- 'betas' and 'Cs' (Sutherland's law constants), or 'Cs','Ts' and 'mus';

- 's0' for a constant entropy, defined by: $s_0 = s_{ref} - R\frac{\gamma}{\gamma-1}ln(T_{ref}) + R\,ln(P_{ref})$ where s_{ref}, T_{ref} and P_{ref} are defined for a reference state.

Computed variables are defined by their CGNS names:

- 'VelocityX', 'VelocityY', 'VelocityZ' for components of the absolute velocity,
- 'VelocityMagnitude' for the absolute velocity magnitude,
- 'Pressure' for the static pressure (requires: gamma),
- 'Temperature' for the static temperature (requires: gamma, rgp),
- 'Enthalpy' for the enthalpy (requires: gamma),
- 'Entropy' for the entropy (requires: gamma, rgp, s0),
- 'Mach' for the Mach number (requires: gamma),
- 'ViscosityMolecular' for the fluid molecular viscosity (requires: gamma, rgp, Ts, mus, Cs),
- 'PressureStagnation' for stagnation pressure(requires: gamma),
- 'TemperatureStagnation' for stagnation temperature (requires: gamma, rgp),
- 'PressureDynamic' for dynamic pressure (requires: gamma).

Example of use:

- Compute variables (array):

```
# - computeVariables (array) -
import Converter as C
import Post as P
import Generator as G

ni = 30; nj = 40
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,2))
c = C.array('ro,rou, rov,row,roE', ni, nj, 2)
c = C.initVars(c, 'ro', 1.)
c = C.initVars(c, 'rou', 1.)
c = C.initVars(c, 'rov', 0.)
c = C.initVars(c, 'row', 0.)
c = C.initVars(c, 'roE', 1.)
m = C.addVars([m, c])

#-------------------------------------------
# Pressure and Mach number extraction
# default values of rgp and gamma are used
#-------------------------------------------
```

(continues on next page)

(continued from previous page)

```
p = P.computeVariables(m, ['Mach', 'Pressure'])
m = C.addVars([m, p])
C.convertArrays2File(m, "out.plt")
```

Note: In the pyTree version, if the variable name is prefixed by 'centers:' then the variable is computed at centers only (e.g. 'centers:Pressure'), and if it is not prefixed, then the variable is computed at nodes.

- Compute variables (pyTree):

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

ni = 30; nj = 40
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,2))
vars = ['Density','MomentumX', 'MomentumY', 'MomentumZ', \
        'EnergyStagnationDensity']
for v in vars: m = C.addVars(m, v)

# Pressure and Mach number extraction
m = P.computeVariables(m, ['Mach', 'Pressure'])
C.convertPyTree2File(m, 'out.cgns')
```

Post.**computeExtraVariable**(*a*, *varName*, *gamma=1.4*, *rgp=287.053*, *Cs=110.4*, *mus=1.76e-5*, *Ts=273.15*)

Compute more advanced variables from conservative variables. 'varName' can be:

- Vorticity,
- VorticityMagnitude,
- QCriterion,
- ShearStress,
- SkinFriction,
- SkinFrictionTangential

The computation of the shear stress requires gamma, rgp, Ts, mus, Cs as input data. In the pyTree version, if a reference state node is defined in the pyTree, then thecorresponding reference constants are used. Otherwise, they must be specified as an argument of the function.

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – Input data
- **varName** (*string*) – variable name (can be preceded by 'nodes:' or 'centers:')

Return type

identical to input

Example of use:

- Extra variables computation (array):

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

a = G.cart( (0,0,0), (1,1,1), (50,50,50) )
a = C.initVars(a, 'Density', 1.)
a = C.initVars(a, 'MomentumX', 1.)
a = C.initVars(a, 'MomentumY', 0.)
a = C.initVars(a, 'MomentumZ', 0.)
a = C.initVars(a, 'EnergyStagnationDensity', 100000.)
m = P.computeExtraVariable(a, 'VorticityMagnitude')
q = P.computeExtraVariable(a, 'QCriterion')
tau = P.computeExtraVariable(a, 'ShearStress')
a = C.node2Center(a)
a = C.addVars([a, m, q, tau])

# Skin friction requires a surface array with shear stress already␣
↪computed
wall = T.subzone(a, (1,1,1), (49,49,1))
skinFriction = P.computeExtraVariable(wall, 'SkinFriction')
skinFrictionTangential = P.computeExtraVariable(wall,
↪'SkinFrictionTangential')
wall = C.addVars([wall, skinFriction, skinFrictionTangential])
C.convertArrays2File([wall], 'out.plt')
```

- Extra variables computation (pyTree):

```
# - computeExtraVariable (pyTree) -
import Generator.PyTree as G
```

(continues on next page)

(continued from previous page)

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

def F(x,y): return x*x + y*y

a = G.cart((0,0,0), (1,1,1), (50,50,50))
a = C.initVars(a, 'Density', 1.)
a = C.initVars(a, 'MomentumX', F, ['CoordinateX', 'CoordinateY'])
a = C.initVars(a, 'MomentumY', 0.)
a = C.initVars(a, 'MomentumZ', 0.)
a = C.initVars(a, 'EnergyStagnationDensity', 100000.)
a = P.computeExtraVariable(a, 'centers:VorticityMagnitude')
a = P.computeExtraVariable(a, 'centers:QCriterion')
a = P.computeExtraVariable(a, 'centers:ShearStress')

b = T.subzone(a, (1,1,1), (50,50,1))
b = P.computeExtraVariable(b, 'centers:SkinFriction')
b = P.computeExtraVariable(b, 'centers:SkinFrictionTangential')

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

Post.PyTree.**computeWallShearStress**(*t*)

Compute the shear stress at wall boundaries provided the velocity gradient is already computed. The problem dimension and the reference state must be provided in t, defining the skin mesh.

Exists also as in place version (_computeWallShearStress) that modifies t and returns None.

The function is only available in the pyTree version.

Parameters

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

Return type

identical to input

Example of use:

- Wall shear stress computation (pyTree):

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

(continues on next page)

(continued from previous page)

```
import Transform.PyTree as T
import Post.PyTree as P

a = G.cart((0,0,0), (1,1,1), (50,50,1))
t = C.newPyTree(['Base',a])
C._addState(t, state='EquationDimension', value=3)
C._addState(t, adim='adim1')
C._initVars(t,'{VelocityX}=0.2*{CoordinateX}**2')
C._initVars(t,'{VelocityY}=0.3*{CoordinateY}*{CoordinateX}')
C._initVars(t,'VelocityZ', 0.)
for var in ['VelocityX','VelocityY','VelocityZ']:
    t = P.computeGrad(t,var)
    t = C.node2Center(t,var)
C._initVars(t,'centers:Density', 1.)
C._initVars(t,'centers:EnergyStagnationDensity', 1.)
C._initVars(t,'centers:Temperature', 1.)
t = P.computeWallShearStress(t)
C.convertPyTree2File(t, 'wall.cgns')
```

Post.**computeGrad**(*a*, *varname*)

Compute the gradient $(\nabla x, \nabla y, \nabla z)$ of a field of name *varname* defined in *a*. The returned field is located at cell centers.

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – Input data
- **varname** (*string*) – variable name (can be preceded by 'nodes:' or 'centers:')

Return type

identical to input

Example of use:

- Gradient of density field (array):

```
# - computeGrad (array) -
import Converter as C
import Post as P
import Generator as G

ni = 1001; nj = 1001; nk = 1
```

(continues on next page)

(continued from previous page)

```
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
m = C.initVars(m, '{ro}= 2*{x}+{x}*{y}')
p = P.computeGrad(m,'ro') # p is defined on centers
p = C.center2Node(p) # back on initial mesh
p = C.addVars([m, p])
C.convertArrays2File([p], 'out.plt')
```

- Gradient of density field (pyTree):

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

ni = 30; nj = 40; nk = 10
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
m = C.initVars(m, '{Density}=2*{CoordinateX}+{CoordinateX}*
↪{CoordinateY}')
m = P.computeGrad(m, 'Density')
C.convertPyTree2File(m, 'out.cgns')
```

Post.**computeGrad2**(*a*, *varname*)

Compute the gradient $(\nabla x, \nabla y, \nabla z)$ at cell centers for a field of name *varname* located at cell centers.

Using Converter.array interface:

```
P.computeGrad2(a, ac, indices=None, BCField=None)
```

a denotes the mesh, *ac* denotes the fields located at centers. indices is a numpy 1D-array of face list, BCField is the corresponding numpy array of face fields. They are used to force a value at some faces before computing the gradient.

Using the pyTree version:

```
P.computeGrad2(a, varname)
```

The variable name must be located at cell centers. Indices and BCFields are automatically extracted from BCDataSet nodes: if a BCDataSet node is defined for a BC of the pyTree, the corresponding face fields are imposed when computing the gradient. If volume has already been computed and volume field is present in tree, it is not recomputed for the gradient computation (only NGON cases up to now).

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – Input data
- **varname** (*string*) – variable name (can be preceded by 'nodes:' or 'centers:')

Return type

identical to input

Example of use:

- Gradient of density field with computeGrad2 (array):

```
# - computeGrad2 (array) -
import Converter as C
import Post as P
import Generator as G

m = G.cartNGon((0,0,0), (1,1,1), (4,4,4))

mc = C.node2Center(m)
mc = C.initVars(mc, '{ro}= 2*{x}+{x}*{y}')
#mc = C.initVars(mc, '{ro}=1.')
mv = C.extractVars(mc, ['ro'])

p = P.computeGrad2(m, mv) # p is defined on centers
p = C.center2Node(p) # back on initial mesh
p = C.addVars([m, p])
C.convertArrays2File([p], 'out.plt')
```

- Gradient of density field with computeGrad2 (pyTree):

```
# - computeGrad2 (pyTree) -
import Converter.PyTree as C
import Post.PyTree as P
import Generator.PyTree as G

m = G.cartNGon((0,0,0), (1,1,1), (4,4,4))
C._initVars(m, '{centers:Density}= 2*{centers:CoordinateX}+
↪{centers:CoordinateY}*{centers:CoordinateZ}')
m = P.computeGrad2(m, 'centers:Density')
C.convertPyTree2File(m, 'out.cgns')
```

Post.**computeGradLSQ**(*a*, *varname*, *dim*)

Compute the gradient $(\nabla x, \nabla y, \nabla z)$ at cell centers for a field of name *varname* located at cell centers. Only supports NGon meshes.

Using the pyTree version:

```
a = P.computeGradLSQ(a, varname, dim)
```

Parameters

- **a** (*[pyTree, base, zone, list of zones]*) – Input data
- **varname** (*string*) – variable name (must be preceded by 'centers:')
- **dim** (*integer*) – dimension (2 for pure 2D, 3 for extruded 2D or 3D)

Return type

identical to input

Example of use:

- Gradient of field F with computeGradLSQ (pyTree):

```
import Converter.PyTree as C
import Post.PyTree as P
import Generator.PyTree as G

a = G.cartNGon((0.,0.,0.), (1./10.,1./10.,1./10.), (11,11,3))

def F(x, y, z): return 3.*x*x + 2.*y*y + z

a = C.initVars(a, 'centers:F', F, ['centers:CoordinateX',
↪'centers:CoordinateY', 'centers:CoordinateZ'])

# dim = 2 (pur 2D), 3 (2D extrudé ou 3D)
dim = 3
a = P.computeGradLSQ(a, 'centers:F', dim)

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

Post.**computeHessian**(*a*, *varname*, *dim*)

Computes the hessian $(\frac{\partial^2 f}{\partial x_i \partial x_j})$ at cell centers for a field of name *varname* located at cell centers. Only supports NGon meshes and 2D geometry.

Using the pyTree version:

```
a = P.computeHessian(a, varname, dim)
```

Parameters

- **a** (*[pyTree, base, zone, list of zones]*) – Input data
- **varname** (*string*) – variable name (must be preceded by 'centers:')
- **dim** (*integer*) – dimension (must be 2 for now)

Return type
identical to input

Example of use:

- Hessian of field F with computeHessian (pyTree):

```
import Converter.PyTree as C
import Post.PyTree as P
import Generator.PyTree as G

a = G.cartNGon((0.,0.,0.), (1./10.,1./10.,1./10.), (11,11,2))

def F(x, y): return 3.*x*x + 2.*y*y

a = C.initVars(a, 'centers:F', F, ['centers:CoordinateX',
↪'centers:CoordinateY'])

# dim = 2 (pur 2D), 3 (2D extrudé ou 3D)
dim = 2
a = P.computeHessian(a, 'centers:F', dim)

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

Post.**computeDiv**(*a*, *varname*)

Compute the divergence $\nabla \cdot (\vec{\bullet})$ of a field defined by its component names ['vectX','vectY','vectZ'] defined in *a*. The returned field is located at cell centers.

Using Converter.array interface:

```
P.computeDiv(a, ['vectX','vectY','vectZ'])
```

Using the pyTree version:

```
P.computeDiv(a, 'vect')
```

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – Input data
- **varname** (*string*) – variable name (can be preceded by 'nodes:' or 'centers:')

Return type

identical to input

Example of use:

- Divergence of a vector field (array) with computeDiv:

```
# - computeDiv (array) -
import Converter as C
import Post as P
import Generator as G

ni = 1001; nj = 1001; nk = 1
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
m = C.initVars(m, '{veloX}= 2*{x}+{x}*{y}')
m = C.initVars(m, '{veloY}= 4.*{y}')
m = C.initVars(m, '{veloZ}= {x}+{z}*{z}')
p = P.computeDiv(m, ['veloX','veloY','veloZ']) # p is defined on␣
↪centers
p = C.center2Node(p) # back to initial mesh
p = C.addVars([m, p])
C.convertArrays2File([p], 'out.plt')
```

- Divergence of a vector field (pyTree) with computeDiv:

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

ni = 30; nj = 40; nk = 10
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
m = C.initVars(m, '{fldX}=2*{CoordinateX}+{CoordinateX}*{CoordinateY}
↪')
m = C.initVars(m, '{fldY}=4*{CoordinateY}')
m = C.initVars(m, '{fldZ}={CoordinateX}+{CoordinateZ}*{CoordinateZ}')
m = P.computeDiv(m, 'fld')
C.convertPyTree2File(m, 'out.cgns')
```

Post.**computeDiv2**(*a*, *varname*)

compute the divergence $\nabla \cdot (\vec{\bullet})$ at cell centers for a vector field defined by its variable names ['vectX','vectY','vectZ'] located at cell centers.

Using Converter.array interface:

```
P.computeDiv2(a, ac, indices=None, BCField=None)
```

a denotes the mesh, *ac* denotes the components of the vector field located at centers. indices is a numpy 1D-array of face list, BCField is the corresponding numpy array of face fields. They are used to force a value at some faces before computing the gradients.

Using the pyTree version:

```
P.computeDiv2(a, 'vect')
P.computeDiv2(a, ['vect1', 'vect2'])
```

The variable name must be located at cell centers. Indices and BCFields are automatically extracted from BCDataSet nodes: if a BCDataSet node is defined for a BC of the pyTree, the corresponding face fields are imposed when computing the gradient.

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – Input data
- **varname** (*[string, list of strings]*) – variable name(s) (can be preceded by 'nodes:' or 'centers:')

Return type

identical to input

Example of use:

- Divergence of a vector field (array) with computeDiv2:

```
# - computeDiv2 (array) -
import Converter as C
import Post as P
import Generator as G
import math

m = G.cartNGon((0,0,0), (1,1,1), (4,4,4))

def Fu(a): return math.cos(a)
def Fv(a): return 4.*a
```

(continues on next page)

(continued from previous page)

```
def Fw(a,b,c): return b*(c**2)

mc = C.node2Center(m)
mc = C.initVars(mc, 'fldX', Fu, ['x'])
mc = C.initVars(mc, 'fldY', Fv, ['y'])
mc = C.initVars(mc, 'fldZ', Fw, ['x', 'y', 'z'])
mv = C.extractVars(mc, ['fldX', 'fldY', 'fldZ'])

p = P.computeDiv2(m, mv) # p is defined on centers
p = C.center2Node(p) # back on initial mesh
p = C.addVars([m, p])
C.convertArrays2File([p], 'out.plt')
```

- Divergence of a vector field (pyTree) with computeDiv2:

```
# - computeDiv2 (pyTree) -
import Converter.PyTree as C
import Post.PyTree as P
import Generator.PyTree as G

m = G.cartNGon((0,0,0), (1,1,1), (4,4,4))
C._initVars(m, '{centers:fldX}= cos({centers:CoordinateX})')
C._initVars(m, '{centers:fldY}= 4.*{centers:CoordinateY}')
C._initVars(m, '{centers:fldZ}= {centers:CoordinateY}*
↪{centers:CoordinateZ}**2.')
m = P.computeDiv2(m, 'centers:fld')
C.convertPyTree2File(m, 'out.cgns')
```

Post.**computeNormGrad**(*a*, *varname*)

Compute the norm of gradient $(\nabla x, \nabla y, \nabla z)$ of a field of name varname defined in a. The returned field 'grad'+varname and is located at cell centers. **(???)**

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – Input data
- **varname** (*string*) – variable name (can be preceded by 'nodes:' or 'centers:')

Return type

identical to input

Example of use:

- Norm of gradient of density (array):

```
# - computeNormGrad (array) -
import Converter as C
import Post as P
import Generator as G

ni = 11; nj = 11; nk = 1
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
m = C.initVars(m, '{ro}= 2*{x}+{x}*{y}')
p = P.computeNormGrad(m,'ro') # p is defined on centers
p = C.center2Node(p) # back on initial mesh
p = C.addVars([m, p])
C.convertArrays2File([p], 'out.plt')
```

- Norm of gradient of density (pyTree):

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

ni = 30; nj = 40; nk = 10
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
m = C.initVars(m, '{Density}=2*{CoordinateX}+{CoordinateX}*
↪{CoordinateY}')
m = P.computeNormGrad(m, 'Density')
C.convertPyTree2File(m, 'out.cgns')
```

Post.**computeCurl**(*a*[, *'vectx'*, *'vecty'*, *'vectz'*])

Compute curl of a 3D vector defined by its variable names ['vectx','vecty','vectz'] in a. The returned field is defined at cell centers for structured grids and elements centers for unstructured grids.

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – Input data
- **vect*** (*string*) – variable name defining the 3D vector

Return type

identical to input

Example of use:

- Curl of momentum field (array):

```
# - computeCurl (array) -
import Converter as C
import Post as P
import Generator as G

def F(x,y,z):
    return 12*y*y + 4

ni = 30; nj = 40; nk = 3
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
m = C.initVars(m,'F1',F,['x','y','z'])
m = C.initVars(m,'F2',0.); m = C.initVars(m,'F3',0.)

varname = ['F1','F2','F3']
p = P.computeCurl(m, varname) # defined on centers
p = C.center2Node(p) # back on init grid
p = C.addVars([m,p])
C.convertArrays2File([p], "out.plt")
```

- Curl of momentum field (pyTree):

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

ni = 30; nj = 40; nk = 3
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
m = C.initVars(m,'{F1}=12*{CoordinateY}*{CoordinateY}+4')
m = C.addVars(m,'F2'); m = C.addVars(m,'F3')

varname = ['F1','F2','F3']
m = P.computeCurl(m, varname)
C.convertPyTree2File(m, 'out.cgns')
```

Post.**computeNormCurl**(*a*[, *'vectx'*, *'vecty'*, *'vectz'*])

Compute the norm of the curl of a 3D vector defined by its variable names ['vectx','vecty','vectz'] in a.

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – Input data
- **vect*** (*string*) – variable name defining the 3D vector

Return type
identical to input

Example of use:

- Norm of the curl of momentum field (array):

```
# - computeNormCurl (array) -
import Converter as C
import Post as P
import Generator as G

def F(x,y,z):
    return 12*y*y + 4

ni = 30; nj = 40; nk = 3
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
m = C.initVars(m,'F1',F,['x','y','z'])
m = C.initVars(m,'F2',0.); m = C.initVars(m,'F3',0.)

varname = ['F1','F2','F3']
p = P.computeNormCurl(m, varname) # defined on centers
p = C.center2Node(p) # back on init grid
p = C.addVars([m,p])
C.convertArrays2File([p], "out.plt")
```

- Norm of the curl of momentum field (pyTree):

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

def F(x,y,z): return 12*y*y + 4

ni = 30; nj = 40; nk = 3
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
m = C.initVars(m,'F1',F,['CoordinateX','CoordinateY','CoordinateZ'])
m = C.addVars(m,'F2'); m = C.addVars(m,'F3')
```

(continues on next page)

(continued from previous page)

```
varname = ['F1','F2','F3']
m = P.computeNormCurl(m, varname)
C.convertPyTree2File(m, 'out.cgns')
```

Post.**computeDiff**(*a*, *varname*)

Compute the difference between neighbouring cells of a scalar field defined by its variable varname in a. The maximum of the absolute difference among all directions is kept.

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – Input data
- **varname** (*string*) – variable name (can be preceded by 'nodes:' or 'centers:')

Return type

identical to input

Example of use:

- Difference of density field (array):

```
# - computeDiff (array) -
import Converter as C
import Post as P
import Generator as G

def F(x):
    if ( x > 5. ): return True
    else : return False

ni = 30; nj = 40; nk = 1
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1.), (ni,nj,nk))
m = C.initVars(m, 'ro', F, ['x'])
p = P.computeDiff(m,'ro')
p = C.addVars([m, p])
C.convertArrays2File([p], 'out.plt')
```

- Difference of density field (pyTree):

```
# - computeDiff (pyTree) -
import Converter.PyTree as C
import Post.PyTree as P
```

(continues on next page)

(continued from previous page)

```
import Generator.PyTree as G

ni = 30; nj = 40; nk = 1
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
m = C.initVars(m, '{Density}=({CoordinateX}>5)*1.')
m = P.computeDiff(m, 'Density')
C.convertPyTree2File(m, 'out.cgns')
```

3.2 Solution selection

Post.**selectCells**(*a*, *F*[, *'var1'*, *'var2'*], *strict=0*, *cleanConnectivity=True*)

Select cells with respect to a given criterion. If strict=0, the cell is selected if at least one of the cell vertices satisfies the criterion. If strict=1, the cell is selected if all the cell vertices satisfy the criterion. The criterion can be defined as a python function returning True (=selected) or False (=not selected):

```
P.selectCells(a, F, ['var1', 'var2'], strict=0)
```

or by a formula:

```
P.selectCells(a, '{x}+{y}>2', strict=0)
```

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – input data
- **F** (*function*) – cells selection criterion
- **var*** (*string*) – arguments of function F
- **strict** (*integer*) – selection mode (0 or 1)
- **cleanConnectivity** (*True or False*) – if True, connectivity is cleaned

Return type

identical to input

Example of use:

- Cell selection in a mesh (array):

```
# - selectCells (array) -
import Converter as C
import Generator as G
import Post as P

a = G.cart( (0,0,0), (1,1,1), (11,11,11) )
def F(x, y, z):
    if (x + 2*y + z > 20.): return True
    else: return False

b = P.selectCells(a, F, ['x', 'y', 'z'])
c = P.selectCells(a, F, ['x', 'y', 'z'], strict=1)
d = P.selectCells(a, '{x}+2*{y}+{z}>20')
e = P.selectCells(a, '({x}>2) & ({y}>2)')
C.convertArrays2File([b,c,d,e], 'out.plt')
```

- Cell selection in a mesh (pyTree):

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

def F(x, y, z):
    if (x + 2*y + z > 20.): return True
    else: return False

a = G.cart( (0,0,0), (1,1,1), (11,11,11) )
a = P.selectCells(a, F, ['CoordinateX', 'CoordinateY', 'CoordinateZ
↪'])
C.convertPyTree2File(a, 'out.cgns')
```

Post.**selectCells2**(*a*, *tag*, *strict=0*)

Select cells according to a field defined by a variable 'tag' (=1 if selected, =0 if not selected). If 'tag' is located at centers, only cells of tag=1 are selected. If 'tag' is located at nodes and 'strict'=0, the cell is selected if at least one of the cell vertices is tag=1. If 'tag' is located at nodes and 'strict'=1, the cell is selected if all the cell vertices is tag=1. In the array version, the tag is an array. In the pyTree version, the tag must be defined in a 'FlowSolution_t' type node located at cell centers or nodes.

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – input data
- **tag** (*string*) – variable name
- **strict** (*integer*) – selection mode (0 or 1)
- **cleanConnectivity** (*True or False*) – if True, connectivity is cleaned

Return type
identical to input

Example of use:

- Cell selection in a mesh with selectCells2 (array):

```
# - selectCells2 (array) -
import Converter as C
import Generator as G
import Post as P

a = G.cart((0,0,0), (1,1,1), (11,11,11))
tag = C.array('tag', 10, 10, 10); tag = C.initVars(tag, 'tag', 1.)
b = P.selectCells2(a, tag)
C.convertArrays2File([b], 'out.plt')
```

- Cell selection in a mesh with selectCells 2 (pyTree):

```
# - selectCells2 (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Post.PyTree as P

a = G.cart((0,0,0), (1,1,1), (11,11,11))
a = C.initVars(a, 'tag', 1.)
b = P.selectCells2(a, 'tag')
C.convertPyTree2File(b, 'out.cgns')
```

Post.**interiorFaces**(*a*, *strict=0*)

Select the interior faces of a mesh. Interior faces are faces with two neighbouring elements. If 'strict' is set to 1, select the interior faces that have only interior nodes.

Parameters

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

- **strict** (*integer*) – selection mode (0 or 1)

Return type
identical to input

Example of use:

- Select interior faces (array):

```
# - interiorFaces (array) -
import Converter as C
import Post as P
import Generator as G

# Get interior faces in broad sense:
# faces with 2 neighbours
a = G.cartTetra((0,0,0), (1,1.,1), (20,2,1))
b = P.interiorFaces(a)
C.convertArrays2File([a,b], 'out1.plt')

# Get interior faces in strict sense:
# faces having only interior nodes
a = G.cartTetra((0,0,0), (1,1.,1), (20,3,1))
b = P.interiorFaces(a,1)
C.convertArrays2File([a,b], 'out2.plt')
```

- Select interior faces (pyTree):

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

a = G.cartTetra((0,0,0), (1,1.,1), (20,20,1))
b = P.interiorFaces(a)
t = C.newPyTree(['Base',1,b])
C.convertPyTree2File(t, 'out.cgns')
```

Post.**exteriorFaces**(*a*, *indices=None*)

Select the exterior faces of a mesh, and return them in a single unstructured zone. If indices=[], the indices of the original exterior faces are returned. For structured grids, indices are the global index containing i faces, then j faces, then k faces, starting from 0. For NGON grids, indices are the NGON face indices, starting from 1.

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – input data
- **indices** (*list of integers*) – indices of original exterior faces

Return type
zone

Example of use:

- Select exterior faces (array):

```
# - exteriorFaces (array) -
import Converter as C
import Post as P
import Generator as G

a = G.cartTetra((0,0,0), (1,1,1), (20,20,20))
indices = []
b = P.exteriorFaces(a, indices=indices)
print(indices)
C.convertArrays2File(b, 'out.plt')
```

- Select exterior faces (pyTree):

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

a = G.cartTetra((0,0,0), (1,1,1), (4,4,6))
b = P.exteriorFaces(a)
C.convertPyTree2File(b, 'out.cgns')
```

Post.**exteriorFacesStructured**(*a*)

Select the exterior faces of a structured mesh as a list of structured meshes.

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

Return type
zone

Example of use:

- Select structured exterior faces (array):

```
# - exteriorFacesStructured (array) -
import Converter as C
import Post as P
import Generator as G

a = G.cart((0,0,0), (1,1,1), (4,4,6))
A = P.exteriorFacesStructured(a)
C.convertArrays2File(A, 'out.plt')
```

- Select structured exterior faces (pyTree):

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

a = G.cart((0,0,0), (1,1,1), (4,4,6))
zones = P.exteriorFacesStructured(a)
C.convertPyTree2File(zones, 'out.cgns')
```

Post.**exteriorElts**(*a*)

Select the exterior elements of a mesh, that is the first border fringe of cells.

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

Return type
identical to input

Example of use:

- Select exterior elements (array):

```
# - exteriorElts (array) -
import Converter as C
import Post as P
import Generator as G

a = G.cartTetra((0,0,0), (1,1,1), (10,10,10))
b = P.exteriorElts(a)
C.convertArrays2File(b, 'out.plt')
```

- Select exterior elements (pyTree):

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

a = G.cartTetra((0,0,0), (1,1,1), (10,10,10))
b = P.exteriorElts(a)
C.convertPyTree2File(b, 'out.cgns')
```

Post.**frontFaces**(*a*, *tag*)

Select faces that are located at the boundary where a tag indicator change from 0 to 1.

param a
input data

type a
[array, list of arrays] or [pyTree, base, zone, list of zones]

param tag
variable name

type tag
string

rtype
zone

Example of use:

- Select a front in a tag (array):

```
# - frontFaces (array) -
import Converter as C
import Generator as G
import Post as P

a = G.cart( (0,0,0), (1,1,1), (11,11,11) )
def F(x, y, z):
    if (x + 2*y + z   > 20.): return 1
    else: return 0
a = C.initVars(a, 'tag', F, ['x', 'y', 'z'])
t = C.extractVars(a, ['tag'])
f = P.frontFaces(a, t)
C.convertArrays2File([a,f], 'out.plt')
```

- Select a front in a tag (pyTree):

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

a = G.cart( (0,0,0), (1,1,1), (11,11,11) )
def F(x, y, z):
    if (x + 2*y + z > 20.): return 1
    else: return 0
a = C.initVars(a, 'tag', F, ['CoordinateX', 'CoordinateY',
↪'CoordinateZ'])
f = P.frontFaces(a, 'tag'); f[0] = 'front'
t = C.newPyTree(['Base']); t[2][1][2] += [a, f]
C.convertPyTree2File(t, 'out.cgns')
```

Post.**sharpEdges**(*A*, *alphaRef=30.*)

Return sharp edges arrays starting from surfaces or contours. Adjacent cells having an angle deviating from more than alphaRef to 180 degrees are considered as sharp.

Parameters

- **A** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – input data
- **alphaRef** (*float*) – split angle

Return type

list of arrays / zones **??**

Example of use:

- Detect sharp edges of a surface (array):

```
# - sharpEdges ( array) -
import Converter as C
import Generator as G
import Post as P
import Transform as T
a1 = G.cart((0.,0.,0.),(1.5,1.,1.),(2,2,1))
a2 = T.rotate(a1,(0.,0.,0.),(0.,1.,0.),100.)
res = P.sharpEdges([a1,a2],alphaRef=45.)
C.convertArrays2File([a1,a2]+res,"out.plt")
```

- Detect sharp edges of a surface (pyTree):

```
# - sharpEdges (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Post.PyTree as P
import Transform.PyTree as T
a1 = G.cart((0.,0.,0.),(1.5,1.,1.),(2,2,1))
a2 = T.rotate(a1,(0.,0.,0.),(0.,1.,0.),100.);a2[0] = 'cart2'
res = P.sharpEdges([a1,a2],alphaRef=45.)
t = C.newPyTree(['Edge',1,'Base',2]); t[2][1][2] += res;␣
↪t[2][2][2]+=[a1,a2]
C.convertPyTree2File(t,"out.cgns")
```

Post.**silhouette**(*A*, *vector=[1., 0., 0.]*)

Return silhouette arrays starting from surfaces or contours, according to a direction vector.

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – input data
- **vector** (*3-tuple of floats*) – direction vector

Return type

identical to input

Example of use:

- Detect silhouette of a surface (array):

```
# - silhouette ( array) -
import Generator as G
import Converter as C
import Post as P

a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 10., (50,1,30))

vector=[1.,0.,0.]
res = P.silhouette([a], vector)

l = [a]+res
C.convertArrays2File(l, 'out.plt')
```

- Detect silhouette of a surface (pyTree):

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

a = G.cylinder((0.,0.,0.), 0.5, 1., 360., 0., 10., (50,1,30))
t = C.newPyTree(['Base']); t[2][1][2].append(a)

vector=[1.,0.,0.]
res = P.silhouette(a, vector)
t[2][1][2] += res
C.convertPyTree2File(t, 'out.cgns')
```

Post.**coarsen**(*a*, *indicName='indic'*, *argqual=0.25*, *tol=1.e-6*)

Coarsen a triangle mesh by providing a coarsening indicator, which is 1 if the element must be coarsened, 0 elsewhere. Triangles are merged by edge contraction, if tagged to be coarsened by indic and if new triangles deviate less than tol to the original triangle. Required mesh quality is controled by argqual: argqual equal to 0.5 corresponds to an equilateral triangle, whereas a value near zero corresponds to a bad triangle shape.

Array version: an indic i-array must be provided, whose dimension ni is equal to the number of elements in the initial triangulation:

```
b = P.coarsen(a, indic, argqual=0.1, tol=1.e6)
```

Parameters

- **a** (*array, list of arrays*) – input data
- **indic** (*i-array*) – tagged element (0 or 1)

Return type
identical to input

PyTree version: indic is stored as a solution located at centers:

```
b = P.coarsen(a, indicName='indic', argqual=0.25, tol=1.e-6)
```

Parameters

- **a** (*pyTree, base, zone, list of zones*) – input data
- **indicName** (*string*) – tag variable name

Return type
identical to input

Example of use:

- Coarsen all cells in a 2D mesh (array):

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

# coarsen all cells of a square
ni = 21; nj = 21; nk = 11
hi = 2./(ni-1); hj = 2./(nj-1); hk = 1./(nk-1)
m = G.cart((0.,0.,0.),(hi,hj,hk), (ni,nj,nk))

hi = hi/2; hj = hj/2; hk = hk/2
m2 = G.cart((0.,0.,0.),(hi,hj,hk), (ni,nj,nk))
m2 = T.subzone(m2,(3,3,6),(m[2]-2,m[3]-2,6))
m2 = T.translate(m2, (0.75,0.75,0.25))
m2 = T.perturbate(m2, 0.51)
tri = G.delaunay(m2)

npts = tri[2].shape[1]
indic = C.array('indic', npts, 1, 1)
indic = C.initVars(indic, 'indic', 1)

sol = P.coarsen(tri, indic, argqual=0.25, tol=1.e6)
C.convertArrays2File([tri, sol], 'out.plt')
```

- Coarsen all cells in a 2D mesh (pyTree):

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

ni = 21; nj = 21; nk = 1
hi = 2./(ni-1); hj = 2./(nj-1)
m = G.cart((0.,0.,0.),(hi,hj,1.), (ni,nj,nk)); m = T.perturbate(m, 0.
↪51)
tri = G.delaunay(m); tri = C.initVars(tri, 'centers:indic', 1.)

sol = P.coarsen(tri, 'indic'); sol[0] = 'coarse'
```

(continues on next page)

(continued from previous page)

```
C.convertPyTree2File([sol,tri], 'out.cgns')
```

Post.**refine**()

Refine a triangle mesh by providing a refinement indicator, which is 1 if the element must be refined, 0 elsewhere.

Array version: an indic i-array must be provided, whose dimension ni is equal to the number of elements in the initial triangulation:

```
b = P.refine(a, indic)
```

PyTree version: indic is stored as a solution located at centers:

```
b = P.refine(a, indicName='indic')
```

Example of use:

- Refine all cells in a 2D mesh (array):

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

# Using indic (linear)
a = G.cartTetra((0,0,0), (1,1,1), (10,10,1))
indic = C.array('indic', a[2].shape[1], 1, 1)
indic = C.initVars(indic, 'indic', 0)
C.setValue(indic, 50, [1])
C.setValue(indic, 49, [1])
a = P.refine(a, indic)
C.convertArrays2File(a, 'out.plt')

# Using butterfly
a = G.cartTetra((0,0,0), (2,1,1), (3,3,3))
a = P.exteriorFaces(a)
#a = C.initVars(a, "z = 0.1*{x}*{x}+0.2*{y}")
for i in range(6):
    a = P.refine(a, w=1./64.)
C.convertArrays2File(a, 'out.plt')
```

- Refine all cells in a 2D mesh (pyTree):

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

# Linear with indicator field
a = G.cartTetra((0,0,0), (1,1,1), (10,10,1))
a = C.initVars(a, 'centers:indic', 1.)
a = P.refine(a, 'indic')
C.convertPyTree2File(a, 'out.cgns')
```

Post.**refine**(*a*, *w=1. / 64.*)

Refine a triangle mesh every where using butterfly interpolation with coefficient w.

Example of use:

- Refine all cells with butterfly interpolation (array):

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

# Refine using butterfly
a = G.cartTetra((0,0,0), (2,1,1), (3,3,3))
a = P.exteriorFaces(a)
for i in range(5):
    a = P.refine(a, w=1./64.)
C.convertArrays2File(a, 'out.plt')
```

- Refine all cells with butterfly interpolation (pyTree):

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

# Refine with butterfly interpolation
a = G.cartTetra((0,0,0), (1,1,1), (10,10,1))
a = P.refine(a, w=1./64.)
C.convertPyTree2File(a, 'out.cgns')
```

Post.**computeIndicatorValue**(*a*, *t*, *varName*)

Compute the indicator value on the unstructured octree mesh a based on the absolute maximum value of a varName field defined in the corresponding structured octree t. In the array version, t is a list of zones, and in the pyTree version, it can be a tree or a base or a list of bases or a zone or a list of zones. Variable varName can be located at nodes or centers. The resulting projected field is stored at centers in the octree mesh.

Example of use:

- Project the maximum value of the indicator field on the octree mesh (array):

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

s = D.circle((0,0,0),1.)
snear = 0.1
o = G.octree([s], [snear], dfar=10., balancing=0)
res = G.octree2Struct(o, vmin=11,merged=0)
vol = G.getVolumeMap(res); res = C.node2Center(res)
val = P.computeIndicatorValue(o,res,vol)
o = C.addVars([o,val])
C.convertArrays2File([o], "out.plt")
```

- Project the maximum value of the indicator field on the octree mesh (pyTree):

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

s = D.circle((0,0,0),1.)
snear = 0.1
o = G.octree([s], [snear], dfar=10., balancing=1)
res = G.octree2Struct(o, vmin=11,merged=1)
res = G.getVolumeMap(res)
o = P.computeIndicatorValue(o,res,'centers:vol')
t = C.newPyTree(['Base']); t[2][1][2] +=[o]
C.convertPyTree2File(t,"out.cgns")
```

Post.**computeIndicatorField**()

compute an indicator field to adapt an octree mesh with respect to the required number of points nbTargetPts, a field, and bodies. If refineFinestLevel=1, the finest level of the octree o is refined. If coarsenCoarsestLevel=1, the coarsest level of the octree o is coarsened provided the balancing is respected.
 This function computes epsInf, epsSup, indicator such that when indicVal < valInf, the octree is coarsened (indicator=-1), when indicVal > valSup, the octree is refined (indicator=1).

For an octree defined in an array o, and the field in indicVal:

```
indicator, valInf, valSup = P.computeIndicatorField(o, indicVal,␣
↪nbTargetPts=-1, bodies=[], refineFinestLevel=1,␣
↪coarsenCoarsestLevel=1)
```

For the pyTree version, the name varname of the field on which is based the indicator must be specified:

```
o, valInf, valSup = P.computeIndicatorField(o, varname, nbTargetPts=-
↪1, bodies=[], refineFinestLevel=1, coarsenCoarsestLevel=1)
```

Example of use:

- Compute the adaptation indicator (array):

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

s = D.circle((0,0,0), 1., N=100); snear = 0.1
o = G.octree([s], [snear], dfar=10., balancing=1)
npts = len(o[1][0])
indicVal = G.getVolumeMap(o)
indicator, valInf, valSup = P.computeIndicatorField(
    o, indicVal, nbTargetPts=2.*npts, bodies=[s])
indicator = C.center2Node(indicator)
o = C.addVars([o, indicator])
C.convertArrays2File(o, "out.plt")
```

- Compute the adaptation indicator (pyTree):

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

(continues on next page)

(continued from previous page)

```
import Geom.PyTree as D
import Post.PyTree as P

#----------------------
# indicateur en centres
#----------------------
s = D.circle((0,0,0), 1., N=100); snear = 0.1
o = G.octree([s], [snear], dfar=10., balancing=1)
npts = o[1][0][0]
o = G.getVolumeMap(o)
o, valInf, valSup = P.computeIndicatorField(o, 'centers:vol',
                                            nbTargetPts=2*npts,␣
↪bodies=[s])
C.convertPyTree2File(o, 'out.cgns')
```

3.3 Solution extraction

Post.**extractPoint**(*A*, *(x, y, z)*, *order=2*, *constraint=40.*, *tol=1.e-6*, *hook=None*)

Extract the field in one or several points, given a solution defined by A. The extracted field(s) is returned as a list of values for each point. If the point (x,y,z) is not interpolable from a grid, then 0 for all fields is returned.

To extract field in several points use:

```
F = P.extractPoint(A, [(x1,y1,z1),(x2,y2,z2)], order=2,␣
↪constraint=40., tol=1.e-6, hook=None)
```

In the pyTree version, extractPoint returns the extracted solution from solutions located at nodes followed by the solution extracted from solutions at centers.

If 'cellN', 'ichim', 'cellnf', 'status', or 'cellNF' variable is defined, it is returned in the last position in the output array. The interpolation order can be 2, 3, or 5.

'constraint' is a thresold for extrapolation to occur. To enable more extrapolation, rise this value.

If some blocks in A define surfaces, a tolerance 'tol' for interpolation cell search can be defined.

A hook can be defined in order to keep in memory the ADT on the interpolation cell search. The hook argument must be a list of hooks, each one being built for each donor zone using the "createHook" function of Converter module with 'extractMesh' argument.

Example of use:

- Extraction in one point (array):

```
# - extractPoint (array) -
import Converter as C
import Generator as G
import Post as P

ni = 10; nj = 10; nk = 10;
a = G.cart((0,0,0), (1./(ni-1),1./(nj-1),1./(nk-1)), (ni,nj,nk))
def F(x,y,z): return x*x*x*x + 2.*y + z*z
a = C.initVars(a, 'F', F, ['x','y','z'])

# Utilisation directe
val = P.extractPoint([a], (0.55, 0.38, 0.12), 2); print(val)

# Utilisation avec un hook
hook = C.createHook([a], function='extractMesh')
val = P.extractPoint([a], (0.55, 0.38, 0.12), 2, hook=hook);␣
↪print(val)
```

- Extraction in one point (pyTree):

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

ni = 10; nj = 10; nk = 10;
a = G.cart((0,0,0), (1./(ni-1),1./(nj-1),1./(nk-1)), (ni,nj,nk))
def F(x,y,z): return x + 2.*y + 3.*z
a = C.initVars(a, 'F', F, ['CoordinateX','CoordinateY','CoordinateZ
↪'])

# Utilisation directe
val = P.extractPoint(a, (0.55, 0.38, 0.12), 2); print(val)

# Utilisation avec un hook
hook = C.createHook(a, function='extractMesh')
val = P.extractPoint(a, (0.55, 0.38, 0.12), 2, hook=[hook]);␣
↪print(val)
```

Post.**extractPlane**(*A*, *(c1, c2, c3, c4)*, *order=2*, *tol=1.e-6*)

slice a solution A with a plane. The extracted solution is interpolated from A. Interpolation order can be 2, 3, or 5 (but the 5th order is very time-consuming for the moment). The best solution is kept. Plane is defined by $c1\ x + c2\ y + c3\ z + c4 = 0$.

Example of use:

- Extraction on a given plane (array):

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

m = G.cylinder((0,0,0), 1, 5, 0., 360., 10., (50,50,50))
m = T.rotate(m , (0,0,0), (1,0,0), 35.)
a = P.extractPlane([m], (0.5, 1., 0., 1), 2)
C.convertArrays2File([m,a], "out.plt")
```

- Extraction on a given plane (pyTree):

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

m = G.cylinder((0,0,0), 1, 5, 0., 360., 10., (50,50,50))
m = T.rotate(m, (0,0,0), (1,0,0), 35.)
m = C.initVars(m,'Density',1); m = C.initVars(m,'centers:cellN',1)
z = P.extractPlane(m, (0.5, 1., 0., 1), 2)
C.convertPyTree2File(z, 'out.cgns')
```

Post.**extractMesh**(*A*, *a*, *order=2*, *extrapOrder=1*, *constraint=40.*, *tol=1.e-6*, *mode='robust'*, *hook=None*)

Interpolate a solution from a set of donor zones defined by A to an extraction zone a. Parameter order can be 2, 3 or 5, meaning that 2nd, 3rd and 5th order interpolations are performed.

Parameter 'constraint'>0 enables to extrapolate from A if interpolation is not possible for some points. Extrapolation order can be 0 or 1 and is defined by extrapOrder.

If mode='robust', extract from the node mesh (solution in centers is first put to nodes, resulting interpolated solution is located in nodes).

If mode='accurate', extract node solution from node mesh and center solution from center mesh (variables don't change location).

The interpolation cell search can be preconditioned if extractMesh is applied several times using the same donor mesh. Parameter hook is only used in 'robust' mode and is a list of ADT (one per donor zone), each of them must be created and deleted by C.createHook and C.freeHook (see Converter module userguide).

Exists also as in place version (_extractMesh) that modifies a and return None.

Example of use:

- Extraction on an extraction zone (array):

```
# - extractMesh (array) -
import Converter as C
import Post as P
import Generator as G

ni = 30; nj = 40; nk = 10
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
m = C.initVars(m, 'ro', 1.)
# Create extraction mesh
a = G.cart((0.,0.,0.), (1., 0.1, 0.1), (20, 20, 1))
# Extract solution on extraction mesh
a2 = P.extractMesh([m], a)
C.convertArrays2File([m,a2], 'out.plt')
```

- Extraction on an extraction zone (pyTree):

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

ni = 30; nj = 40; nk = 10
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk))
C._initVars(m, 'Density', 2.)
C._initVars(m, 'centers:cellN', 1)

# Extraction mesh
a = G.cart((0.,0.,0.5), (1., 0.1, 1.), (20, 20, 1)); a[0] =
↪'extraction'

# Extract solution on extraction mesh
```

(continues on next page)

(continued from previous page)

```
a = P.extractMesh(m, a)
t = C.newPyTree(['Solution', 3, 'Extraction', 2])
t[2][1][2].append(m); t[2][2][2] += [a]
C.convertPyTree2File(a, 'out.cgns')
```

Post.**projectCloudSolution**(*pts*, *t*, *dim=3*)

Project the solution by a Least-Square Interpolation defined on a set of points pts defined as a 'NODE' zone to a body defined by a 'TRI' mesh in 3D and 'BAR' mesh in 2D.

Example of use:

- projectCloudSolution (array):

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

a = D.sphere((0,0,0),1.,N=20)
a = C.convertArray2Tetra(a); a = G.close(a)
b = D.sphere6((0,0,0),1.,N=15)
b = C.convertArray2Tetra(b); b = T.join(b)
pts = C.convertArray2Node(b)
pts = C.initVars(pts,'{F}={x}*{y}')
a = C.initVars(a,'F', 0.)
a = P.projectCloudSolution(pts,a, loc='nodes')
C.convertArrays2File(a,"out.plt")
```

- projectCloudSolution (pyTree):

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

a = D.sphere((0,0,0),1.,N=20)
a = C.convertArray2Tetra(a); a = G.close(a)
b = D.sphere6((0,0,0),1.,N=15)
```

(continues on next page)

(continued from previous page)

```
b = C.convertArray2Tetra(b); b = T.join(b)
pts = C.convertArray2Node(b)
C._initVars(pts,'{F}={CoordinateX}*{CoordinateY}')
C._initVars(a,'F', 0.)
a = P.projectCloudSolution(pts,a)
C.convertPyTree2File(a, "out.cgns")
```

Post.**zipper**(*A*, *options=[]*)

Build an unstructured unique surface mesh, given a list of structured overlapping surface grids A. Cell nature field is used to find blanked (0) and interpolated (2) cells.

The options argument is a list of arguments such as ["argName", argValue]. Option names can be:

- 'overlapTol' for tolerance required between two overlapping grids : if the projection distance between them is under this value then the grids are considered to be overset. Default value is 1.e-5.
- For some cases, 'matchTol' can be set to modify the matching boundaries tolerance. Default value is set 1e-6.

In most cases, one needn't modify this parameter.

Example of use:

- Zipping of an overset surface (array):

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

m1 = G.cylinder((0,0,0), 1, 5, 0., 360., 10., (50,50,1))
m1 = C.initVars(m1, 'cellN', 1.)

# Set cellN = 2 (interpolated points) to boundary
s = T.subzone(m1, (1,m1[3],1),(m1[2],m1[3],m1[4]))
s = C.initVars(s, 'cellN', 2)
m1 = T.patch(m1, s, (1,m1[3],1))
s = T.subzone(m1, (1,1,1),(m1[2],1,m1[4]))
s = C.initVars(s, 'cellN', 2)
m1 = T.patch(m1, s, (1,1,1))
```

(continues on next page)

(continued from previous page)

```
ni = 30; nj = 40
m2 = G.cart((0,0,0), (10./(ni-1),10./(nj-1),-1), (ni,nj,1))
m2 = C.initVars(m2, 'cellN', 1.)

array = P.zipper([m1,m2],[])
C.convertArrays2File([array], 'out.plt')
```

- Zipping of an overset surface (pyTree):

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

# cylindre
ni = 30; nj = 40; nk = 1
m1 = G.cylinder((0,0,0), 1, 5, 0., 360., 10., (ni,nj,nk))
m1 = C.addVars(m1, 'Density'); m1 = C.initVars(m1,'cellN',1)

# Set cellN = 2 (interpolated points) to boundary
s = T.subzone(m1, (1,nj,1),(ni,nj,nk))
s = C.initVars(s, 'cellN', 2)
m1 = T.patch(m1, s, (1,nj,1))
s = T.subzone(m1, (1,1,1),(ni,1,nk))
s = C.initVars(s, 'cellN', 2)
m1 = T.patch(m1, s, (1,1,1))

# carre
ni = 30; nj = 40
m2 = G.cart((0,0,0), (10./(ni-1),10./(nj-1),-1), (ni,nj,1))
m2 = C.initVars(m2, 'Density', 1.2); m2 = C.initVars(m2, 'cellN', 1.)

t = C.newPyTree(['Base',2]); t[2][1][2] += [m1, m2]
z = P.zipper(t); z[0] = 'zipper'; t[2][1][2].append(z)
C.convertPyTree2File(t, 'out.cgns')
```

Post.**usurp**(*A*)

This function computes a "ratio" field for structured overlapping surfaces. The ratio field is located at cell centers. In case of no overset, ratio are set to 1, otherwise ratio represents the percentage of overlap of a cell by another mesh. The finest cells have priority. All surfaces must be oriented in the same way.

When using the array interface:

```
C = P.usurp(A, B)
```

the input arrays are a list of grid arrays A, defining nodes coordinates and a corresponding list of arrays defining the chimera nature of cells at cell centers B. Blanked cells must be flagged by a null value. Other values are equally considered as computed or interpolated cells.

When using the pyTree interface:

```
C = P.usurp(A)
```

chimera cell nature field must be defined as a center field in A.

Warning: normal of surfaces grids defined by A must be oriented in the same direction.

Example of use:

- Ratio generation for the surface elements (array):

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

cyln = []
a1 = G.cylinder((0,0,0), 0, 2, 360, 0, 1., (100,2,10))
a1 = T.subzone(a1, (1,2,1), (a1[2],2,a1[4]))
cyln.append(a1)
#
a2 = G.cylinder((0,0,0), 0, 2, 90, 0, 0.5, (10,2,10))
a2 = T.translate(a2, (0,0,0.2))
a2 = T.subzone(a2, (1,2,1), (a2[2],2,a2[4]))
cyln.append(a2)
#
c1 = cyln[0]
ib1 = C.array('cellN', c1[2]-1, c1[3], c1[4]-1)
ib1 = C.initVars(ib1,'cellN', 1)
ib1[1][0,586] = 0.
#
c2 = cyln[1]
ib2 = C.array('cellN', c2[2]-1, c2[3], c2[4]-1)
ib2 = C.initVars(ib2, 'cellN', 1)

ibc = [ib1, ib2]; r = P.usurp(cyln, ibc)
cylc = C.node2Center(cyln)
```

(continues on next page)

(continued from previous page)

```
out = []
l = len(cylc)
for i in range(l) :
    out.append(C.addVars([cylc[i], ibc[i]]))

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

- Ratio generation for the surface elements (pyTree):

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

a1 = G.cylinder((0,0,0), 0, 2, 360, 0, 1., (100,2,10))
a1 = T.subzone(a1, (1,2,1), (100,2,10)); a1[0]='cyl1'

a2 = G.cylinder((0,0,0), 0, 2, 90, 0, 0.5, (10,2,10))
a2 = T.translate(a2, (0,0,0.2))
a2 = T.subzone(a2, (1,2,1),(10,2,10)); a2[0]='cyl2'

a1 = C.initVars(a1, 'centers:cellN',1.)
a2 = C.initVars(a2, 'centers:cellN',1.)
t = C.newPyTree(['Base',2]); t[2][1][2] += [a1, a2]
t = P.usurp(t)
C.convertPyTree2File(t, 'out.cgns')
```

Post.Probe.**Probe**(*fileName*, *t=None*, *X=(x, y, z)*, *ind=None*, *blockName=None*, *tPermeable=None*, *fields=None*, *append=True*, *bufferSize=100*)

Create a probe. 3 modes are possible :

- mode 0 : if t and (x,y,z) are provided, the probe will extract given fields from t at single position (x,y,z).
- mode 1 : if t and (ind, blockName) are provided, the probe will extract given fields from block of t at single index ind of blockName.
- mode 2 : if t, (x,y,z) and ind are not provided, the probe will store the zones given at extract.
- mode 3 : if tPermeable is provided, the probe will interpolate on tPermeable.

Result is periodically flush to file when buffer size exceeds bufferSize.

Parameters

- **fileName** (*string*) – name of file to dump to
- **t** (*pyTree*) – pyTree containing solution
- **(x,y,z)** (*tuple of 3 floats*) – position of single probe (mode 0)
- **ind** (*integer*) – index of single probe in blockName (mode 1)
- **blockName** (*string*) – name of block containing probe (mode 1)
- **tPermeable** (*pyTree, zone or list of zones*) – surface to interpolate to.
- **fields** (*list of strings or None*) – list of fields to extract
- **append** (*Boolean*) – if True, append result to existing file
- **bufferSize** (*int*) – size of internal buffer

Return type

probe instance

Example of use:

- Probe extraction (pyTree):

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

# test case
a = G.cartRx((0,0,0), (1,1,1), (30,30,30), (5,5,5), depth=0,␣
↪addCellN=False)
C._initVars(a, '{centers:F} = {centers:CoordinateX}')
t = C.newPyTree(['Base', a])

# create a probe
p1 = Probe.Probe('probe1.cgns', t, X=(10.,10.,10.), fields=[
↪'centers:F'], append=False)
p1.printInfo()
for i in range(110):
    time = 0.1*i
    C._initVars(t, '{centers:F} = {centers:CoordinateX}+10.*sin(%20.
↪16g)'%time)
    p1.extract(t, time=time)
p1.flush()
```

(continues on next page)

(continued from previous page)

```
# reread probe from file
p1 = Probe.Probe('probe1.cgns')
out = p1.read()
C.convertPyTree2File(out, 'out.cgns')
```

Post.Probe.Probe.**extract**(*t*, *time=0.*)

Extract probe at given time from t.

Parameters

- **t** (*pyTree*) – pyTree containing solution
- **time** (*float*) – extraction time

Post.Probe.Probe.**flush**()

Force probe flush to disk.

Post.Probe.Probe.**read**(*cont=None*, *ind=None*, *probeName=None*)

Read all data stored in probe file and return a zone. Can be used in two ways:

- cont: extract the given time container (all probe points)
- ind, probeName: extract the given index, zone (all times)

Parameters

- **cont** (*integer*) – number of time container
- **ind** (*integer*) – index to extract
- **probeName** (*string*) – name of probe zone to extract

3.4 Streams

Post.**streamLine**(*A*, *(x0, y0, z0)*[, *'v1'*, *'v2*, *'v3'*], *N=2000*, *dir=2*)

Compute the stream line with N points starting from point (x0,y0,z0), given a solution A and a vector defined by 3 variables ['v1','v2,'v3']. Parameter 'dir' can be set to 1 (streamline follows velocity), -1 (streamline follows -velocity), or 2 (streamline expands in both directions). The output yields the set of N extracted points on the streamline, and the input fields at these

points. The streamline computation stops when the current point is not interpolable from the input grids.

Example of use:

- Streamline extraction (array):

```
# - streamLine (array) -
import Converter as C
import Post as P
import Generator as G
import math as M

ni = 30; nj = 40
m1 = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,2))
m2 = G.cart((5.5,0,0), (9./(ni-1),9./(nj-1),1), (ni,nj,2))

def F(x): return M.cos(x)

m = [m1,m2]
m = C.initVars(m, 'rou', 1.)
m = C.initVars(m, 'rov', F, ['x'])
m = C.initVars(m, 'row', 0.)
x0=0.1; y0=5.; z0=0.5; p = P.streamLine(m, (x0,y0,z0),['rou','rov',
↪'row'])
C.convertArrays2File(m+[p], 'out.plt')
```

- Streamline extraction (pyTree):

```
# - streamLine (pyTree) -
import Converter.PyTree as C
import Post.PyTree as P
import Generator.PyTree as G
import math as M

def F(x): return M.cos(x)
ni = 30; nj = 40
m1 = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,2))
m2 = G.cart((5.5,0,0), (9./(ni-1),9./(nj-1),1), (ni,nj,2))
t = C.newPyTree(['Base','StreamR']); t[2][1][2] = [m1,m2]
t = C.initVars(t, 'vx', 1.)
t = C.initVars(t, 'vy', F, ['CoordinateX'])
t = C.initVars(t, 'vz', 0.)
x0=0.1; y0=5.; z0=0.5
```

(continues on next page)

(continued from previous page)

```
p = P.streamLine(t, (x0,y0,z0),['vx','vy','vz'])
C.convertPyTree2File(p, "out.cgns")
```

Post.**streamRibbon**(*A*, *(x0, y0, z0)*, *(nx, ny, nz)*[, *'v1'*, *'v2'*, *'v3'*], *N=2000*, *dir=2*)

0 Compute the stream ribbon starting from point (x0,y0,z0), of width and direction given by the vector (nx,ny,nz).

This vector must be roughly orthogonal to the vector ['v1', 'v2', 'v3'] at point (x0,y0,z0). The output yields the set of N extracted points on the stream ribbon, and the input fields at these points. The stream ribbon computation stops when the current point is not interpolable from the input grids.

Example of use:

- Stream ribbon extraction (array):

```
# - streamRibbon (array) -
import Converter as C
import Post as P
import Generator as G
import math as M

ni = 30; nj = 40
def F(x): return M.cos(x)

m1 = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,2))
m2 = G.cart((5.5,0,0), (9./(ni-1),9./(nj-1),1), (ni,nj,2))
m = [m1,m2]
m = C.initVars(m, 'rou', 1.)
m = C.initVars(m, 'rov', F, ['x'])
m = C.initVars(m, 'row', 0.)
x0=0.1; y0=5.; z0=0.5; p = P.streamRibbon(m, (x0,y0,z0),(0.,0.2,0.),[
↪'rou','rov','row'])
C.convertArrays2File(m+[p], 'out.plt')
```

- Stream ribbon extraction (pyTree):

```
# - streamRibbon (pyTree) -
import Converter.PyTree as C
import Post.PyTree as P
import Generator.PyTree as G
import math as M
```

(continues on next page)

(continued from previous page)

```
def F(x): return M.cos(x)
ni = 30; nj = 40
m1 = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,2))
m2 = G.cart((5.5,0,0), (9./(ni-1),9./(nj-1),1), (ni,nj,2))
t = C.newPyTree(['Base','StreamR']); t[2][1][2] = [m1,m2]
t = C.initVars(t, 'vx', 1.)
t = C.initVars(t, 'vy', F, ['CoordinateX'])
t = C.initVars(t, 'vz', 0.)
x0=0.1; y0=5.; z0=0.5
p = P.streamRibbon(t, (x0,y0,z0),(0.,0.2,0.),['vx','vy','vz'])
C.convertPyTree2File(p, "out.cgns")
```

Post.**streamSurf**(*A*, *c*[, *'v1'*, *'v2*, *'v3'*], *N=2000*, *dir=1*)

Compute the stream surface starting from a BAR array c.

Example of use:

- Stream surface extraction (array):

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

ni = 30; nj = 40

# Node mesh
m1 = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,5))
m2 = G.cart((5.5,0,0), (9./(ni-1),9./(nj-1),1), (ni,nj,5))
b = D.line((0.1,5.,0.1), (0.1,5.,3.9), N=10)
b = C.convertArray2Tetra(b)
m = [m1,m2]
def F(x): return M.cos(x)

m = C.initVars(m, 'rou', 1.)
m = C.initVars(m, 'rov', F, ['x'])
m = C.initVars(m, 'row', 0.)
p = P.streamSurf(m, b,['rou','rov','row'])
C.convertArrays2File(m+[p], 'out.plt')
```

- Stream surface extraction (pyTree):

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

ni = 30; nj = 40; nk = 5
m1 = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,nk)); m1[0] =
↪'cart1'
m2 = G.cart((5.5,0,0), (9./(ni-1),9./(nj-1),1), (ni,nj,nk)); m2[0] =
↪'cart2'
b = D.line((0.1,5.,0.1), (0.1,5.,3.9), N=10)
b = C.convertArray2Tetra(b)

t = C.newPyTree(['Base',m1,m2])
t = C.initVars(t, 'vx', 1.)
t = C.initVars(t, '{vy}=cos({CoordinateX})')
t = C.initVars(t, 'vz', 0.)
x0=0.1; y0=5.; z0=0.5
p = P.streamSurf(t, b, ['vx','vy','vz'])
C.convertPyTree2File(p, "out.cgns")
```

3.5 Isos

`Post.isoLine`(*A*, *field*, *val*)

Compute an isoline correponding to value val of field.

Example of use:

- Isoline computation (array):

```
# - isoLine (array) -
import Post as P
import Converter as C
import Generator as G

def F(x, y): return x*x+y*y

a = G.cartTetra((0,0,0), (1,1,1), (10,10,1))
a = C.initVars(a, 'field', F, ['x','y'])
```

(continues on next page)

(continued from previous page)

```
isos = []
min = C.getMinValue(a, 'field')
max = C.getMaxValue(a, 'field')
for v in range(20):
    value = min + (max-min)/18.*v
    try:
        i = P.isoLine(a, 'field', value)
        if i != []: isos.append(i)
    except: pass
C.convertArrays2File([a]+isos, 'out.plt')
```

- Isoline (pyTree):

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

def F(x, y): return x*x+y*y

a = G.cartTetra( (0,0,0), (1,1,1), (10,10,1))
a = C.initVars(a, 'field', F, ['CoordinateX','CoordinateY'])

isos = []
min = C.getMinValue(a, 'field')
max = C.getMaxValue(a, 'field')
for v in range(20):
    value = min + (max-min)/18.*v
    try:
        i = P.isoLine(a, 'field', value)
        isos.append(i)
    except: pass
t = C.newPyTree(['Base',3,'ISOS',1])
t[2][1][2].append(a)
t[2][2][2] += isos
C.convertPyTree2File(t, 'out.cgns')
```

Post.**isoSurf**(*a*, *field*, *val*, *vars=None*, *split='simple'*)

Compute an isosurface corresponding to value val of field (using marching tetrahedra). Resulting solution is always located in nodes. Return a list of two zones (one TRI and one BAR, if relevant). If vars (for ex: ['centers:F', 'G']) is given, extract only given variables.

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – input data
- **field** (*string*) – field name used in iso computation
- **val** (*float*) – value of field for extraction
- **vars** (*list of strings*) – list of variable names you want to see on final iso-surface
- **split** (*string*) – 'simple' or 'withBarycenters', used in decomposing a in tetra (if needed)

Example of use:

- Isosurface extraction by marching tetra (array):

```
# - isoSurf (array) -
import Post as P
import Converter as C
import Generator as G

a = G.cartTetra((-20,-20,-20), (0.25,0.25,0.5), (100,100,50))
a = C.initVars(a, '{field}={x}*{x}+{y}*{y}+{z}')
iso = P.isoSurf(a, 'field', value=5.)
C.convertArrays2File(iso, 'out.plt')
```

- Isosurface extraction by marching tetra (pyTree):

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

a = G.cartTetra((-20,-20,-20), (0.5,0.5,0.5), (50,50,50))
a = C.initVars(a, '{field}={CoordinateX}*{CoordinateX}+{CoordinateY}*
↪{CoordinateY}+{CoordinateZ}')

iso = P.isoSurf(a, 'field', value=5.)
C.convertPyTree2File(iso, 'out.cgns')
```

Post.**isoSurfMC**(*a*, *field*, *val*, *vars=None*, *split='simple'*)

Compute an isosurface corresponding to value val of field (using marching cubes). Resulting solution is always located in nodes. If vars (for ex: ['centers:F', 'G']) is given, extract only given variables.

Parameters

- **a** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – input data
- **field** (*string*) – field name used in iso computation
- **val** (*float*) – value of field for extraction
- **vars** (*list of strings*) – list of variable names you want to see on final iso-surface
- **split** (*string*) – 'simple' or 'withBarycenters', used in decomposing a in tetra (if needed)

Returns

a list of isosurface (one per original zones)

Return type

list of arrays or list of zones

Example of use:

- Isosurface by marching cube (array):

```
# - isoSurfMC (array) -
import Post as P
import Converter as C
import Generator as G

a = G.cartHexa((-20,-20,-20), (0.25,0.25,0.5), (100,100,50))
a = C.initVars(a, '{field}={x}*{x}+{y}*{y}+{z}')
iso = P.isoSurfMC(a, 'field', value=5.)
C.convertArrays2File(iso, 'out.plt')
```

- Isosurface by marching cube (pyTree):

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

a = G.cartHexa((-20,-20,-20), (0.5,0.5,0.5), (50,50,50))
a = C.initVars(a, '{field}={CoordinateX}*{CoordinateX}+{CoordinateY}*
↪{CoordinateY}+{CoordinateZ}')

iso = P.isoSurfMC(a, 'field', value=5.)
C.convertPyTree2File(iso, 'out.cgns')
```

3.6 Solution integration

For all integration functions, the interface is different when using Converter arrays interface or pyTree interface. For arrays, fields must be input separately, for pyTree, they must be defined in each zone.

Post.**integ**(*A*, *var='F'*)

Compute the integral $\int F.dS$ of a scalar field (whose name is in var string) over the geometry defined by arrays containing the coordinates + field (+ an optional "ratio" field). Solution and ratio can be located at nodes or at centers.

For array interface:

```
res = P.integ([coord], [field], [ratio]=[])
```

For pyTree interface, the variable to be integrated can be specified. If no variable is specified, all the fields located at nodes and centers are integrated:

```
res = P.integ(A, var='F')
```

param A
input data

type A
[array, list of arrays] or [pyTree, base, zone, list of zones]

param var
field name

type var
string

return
the result of integration

rtype
a list of 1 float

Exists also as parallel distributed version (P.Mpi.integ).

Example of use:

- Scalar integration (array):

```
# - integ (array) -
import Converter as C
import Generator as G
import Post as P

# Node mesh
ni = 30; nj = 40
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,1))

# Field in centers
c = C.array('vx', ni-1, nj-1, 1); c = C.initVars(c, 'vx', 1.)
resc = P.integ([m], [c], [])[0]; print(resc)

# Field in nodes
cn = C.array('vx', ni, nj, 1); cn = C.initVars(cn, 'vx', 1.)
resn = P.integ([m], [cn], [])[0]; print(resn)
```

- Scalar integration (pyTree):

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

ni = 30; nj = 40
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,1))
C._initVars(m, 'vx', 1.); C._initVars(m, 'ratio', 1.)
resn = P.integ(m, 'vx'); print(resn)
#>> [99.99999999999989]
```

Post.**integNorm**(*A*, *var='F'*)

Compute the integral $\int F.\vec{n}.dS$ of a scalar field times the surface normal over the geometry defined by coord. For array interface:

```
P.integNorm([coord], [field], [ratio]=[])
```

For pyTree interface, the variable to be integrated can be specified. If no variable is specified, all the fields located at nodes and centers are integrated:

```
P.integNorm(A, var='F')
```

Parameters

- **A** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – input data
- **var** (*string*) – field name

Returns
the result of integration

Return type
double list of 3 floats

Exists also as parallel distributed version (P.Mpi.integNorm).

Example of use:

- Integration dot the surface normal (array):

```
# - integNorm (array) -
import Converter as C
import Generator as G
import Post as P

# Node mesh and field
m = G.cartTetra((0.,0.,0.), (0.1,0.1,0.2), (10,10,1))
c1 = C.array('ro', 100, 162, 'TRI')
c = C.initVars(c1, 'ro', 1.)
res = P.integNorm([m], [c], []); print('res1 = %f'%res)

# Node mesh
ni = 30; nj = 40
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,1))

# Centers field
c1 = C.array('vx', ni-1, nj-1, 1)
c = C.initVars(c1, 'vx', 1.)

# Integration
res = P.integNorm([m], [c], []); print('res2 = %f'%res)

# Node field
c1 = C.array('vx, vy', ni, nj, 1)
cn = C.initVars(c1, 'vx', 1.)
cn = C.initVars(c1, 'vy', 1.)
resn = P.integNorm([m], [cn], []); print('res3 = %f'%resn)
```

- Integration dot the surface normal (pyTree):

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

# Node mesh
m = G.cartTetra((0.,0.,0.), (0.1,0.1,0.2), (10,10,1))
m = C.initVars(m, 'Density', 1.)
t = C.newPyTree(['Base', m])
res = P.integNorm(t, 'Density'); print(res)
#>> [[0.0, 0.0, 0.8099999999999997]]
```

Post.**integNormProduct**(*A*, *vector=['vx', 'vy', 'vz']*)

Compute the integral $\int \vec{V} \times \vec{n}.dS$ of a vector field times the surface normal over the geometry defined by coord. The input field must have 3 variables. For array interface, field must be a vector field:

```
res = P.integNormProduct([coord], [field], [ratio]=[])
```

For pyTree interface, the vector field to be integrated must be specified:

```
res = P.integNormProduct(A, vector=['vx','vy','vz'])
```

Exists also as parallel distributed version (P.Mpi.integNormProduct).

Example of use:

- Integration cross the surface normal (array):

```
# - integNormProduct (array) -
import Converter as C
import Generator as G
import Post as P

# Maillage et champs non structure, en noeuds
m = G.cartTetra((0.,0.,0.), (0.1,0.1,0.2), (10,10,1))
c = C.array('vx,vy,vz', m[1].shape[1], m[2].shape[1], 'TRI')
c = C.initVars(c, 'vx,vy,vz', 1.)
res = P.integNormProduct([m], [c], []); print(res)

# Maillage en noeuds
ni = 30; nj = 40
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,1))
```

(continues on next page)

(continued from previous page)

```
# Champ a integrer en centres
c = C.array('vx,vy,vz', ni-1, nj-1, 1)
c = C.initVars(c, 'vx,vy,vz', 1.)

# Integration de chaque champ
res = P.integNormProduct([m], [c], []); print(res)

# Champ a integrer en noeuds
cn = C.array('vx,vy,vz', ni, nj, 1)
cn = C.initVars(cn, 'vx,vy,vz', 1.)
resn = P.integNormProduct([m], [cn], []); print(resn)
```

- Integration cross the surface normal (pyTree):

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

ni = 30; nj = 40
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,1))
m = C.initVars(m,'MomentumX',1.)
m = C.initVars(m,'MomentumY',1.)
m = C.initVars(m,'MomentumZ',1.)
res = P.integNormProduct(m,['MomentumX','MomentumY','MomentumZ']);␣
↪print(res)
#>> 100.0
```

Post.**integMoment**(*A*, *center=(0., 0., 0.)*, *vector=['vx', 'vy', 'vz']*)

Compute the integral $\int \vec{CM} \times \vec{V}.dS$ of a moment over the geometry defined by coord. The input field must have 3 variables. center=(cx,cy,cz) are the center coordinates.

For array interface:

```
res = P.integMoment([coord], [field], [ratio]=[], center=(0.,0.,0.))
```

For pyTree interface, the vector of variables to be integrated must be specified:

```
res = P.integMoment(A, center=(0.,0.,0.), vector=['vx','vy','vz'])
```

Exists also as parallel distributed version (P.Mpi.integMoment).

Parameters

- **A** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – input data
- **vector** (*list of 3 strings*) – list of vector field names

Returns
the result of integration

Return type
a list of 3 floats

Example of use:

- Moment integration (array):

```
# - integMoment (array) -
import Converter as C
import Generator as G
import Post as P

# Maillage et champs non structure, en noeuds
m = G.cartTetra((0.,0.,0.), (0.1,0.1,0.2), (10,10,1))
c = C.array('vx,vy,vz', 100, 162, 'TRI')
c = C.initVars(c, 'vx,vy,vz', 1.)
res = P.integMoment([m], [c], [], (5.,5., 0.)); print(res)

# Maillage en noeuds
ni = 30; nj = 40
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,1))
C.convertArrays2File([m], "new.plt", "bin_tp")

# Champ a integrer en centres
c = C.array('vx,vy,vz', ni-1, nj-1, 1)
c = C.initVars(c, 'vx,vy,vz', 1.)

# Integration de chaque champ
res = P.integMoment([m], [c], [], (5.,5., 0.) ); print(res)

# Champ a integrer en noeuds
cn = C.array('vx,vy,vz', ni, nj, 1)
cn = C.initVars(cn, 'vx,vy,vz', 1.)
resn = P.integMoment([m], [cn], [], (5.,5., 0.)); print(resn)
```

- Moment integration (pyTree):

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

m = G.cartTetra((0.,0.,0.), (0.1,0.1,0.2), (10,10,1))
C._initVars(m,'vx',1.); C._initVars(m,'vy',0.); C._initVars(m,'vz',0.
↪)
res = P.integMoment(m, center=(5.,5., 0.), vector=['vx','vy','vz']);␣
↪print(res)
#>> [0.0, 0.0, 3.6854999999999976]
```

Post.**integMomentNorm**(*A*, *center=(cx, cy, cz)*, *var='F'*)

Compute the integral $\int \vec{CM} \times F.\vec{n}.dS$ of a moment over the geometry defined by coord, taking into account the surface normal. The input field is a scalar.

For array interface:

```
res = P.integMomentNorm([coord], [field], [ratio]=[], center=(cx,cy,
↪cz))
```

For pyTree interface, the variable to be integrated can be specified. If no variable is specified, all the fields located at nodes and centers are integrated:

```
res = P.integMomentNorm(A, center=(cx,cy,cz), var='F')
```

Parameters

- **A** (*[array, list of arrays] or [pyTree, base, zone, list of zones]*) – input data
- **var** (*string*) – field name

Returns
the result of integration

Return type
a list of 3 floats

Exists also as parallel distributed version (P.Mpi.integMomentNorm).

Example of use:

- Moment integration with normal (array):

```
# - integMomentNorm (array) -
import Converter as C
import Generator as G
import Post as P

# Maillage et champs non structure, en noeuds
m = G.cartTetra((0.,0.,0.), (0.1,0.1,0.2), (10,10,1))
c = C.array('ro', 100, 162, 'TRI')
c = C.initVars(c, 'ro', 1.)
res = P.integMomentNorm([m], [c], [], (5.,5., 0.)); print(res)

# Maillage en noeuds
ni = 30; nj = 40
m = G.cart((0,0,0), (10./(ni-1),10./(nj-1),1), (ni,nj,1))

# Champ a integrer en centres
c = C.array('v', ni-1, nj-1, 1)
c = C.initVars(c, 'v', 1.)

# Integration de chaque champ
res = P.integMomentNorm([m], [c], [], (5.,5., 0.) ); print(res)

# Champ a integrer en noeuds
cn = C.array('v', ni, nj, 1)
cn = C.initVars(cn, 'v', 1.)
resn = P.integMomentNorm([m], [cn], [], (5.,5., 0.)); print(resn)
```

- Moment integration with normal (pyTree):

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

m = G.cartTetra((0.,0.,0.), (0.1,0.1,0.2), (10,10,1))
m = C.initVars(m, 'Density', 1.)
res = P.integMomentNorm(m, var='Density', center=(5.,5.,0.));␣
↪print(res)
#>> [[-3.685500000000002, 3.6855, 0.0]]
```