Post.IBM Documentation

Release 3.7

/ELSA/MU-10019/V3.7

Oct 19, 2023

CONTENTS

Specific post-processing for immersed boundaries (IB).

These functions work with a solution tree “t”, a geometry tree “tb”, and/or a connectivity tree “tc”.

CHAPTER

ONE

LIST OF FUNCTIONS

– Post-processing for IB

Post.IBM.extractIBMWallFields(tc[, tb, ...])	Extracts the flow field stored at IBM points onto the surface.
Post.IBM.extractShearStress(ts)	Computes the shear stress on the surface.
Post.IBM.extractLocalPressureGradients(ts)	Computes the pressure gradients in the wall normal/tangent direction.
Post.IBM.extractYplusIP(tc)	Computes yplus values at image points and store them in the tc.
Post.IBM.extractPressureHO(tc[, extractDensity])	Extrapolates the wall pressure (1st order) at the immersed boundaries and stores the solution in the tc.
Post.IBM.extractPressureHO2(tc[, extractDensity])	Extrapolates the wall pressure (2nd order) at the immersed boundaries and stores the solution in the tc.
Post.IBM.extractConvectiveTerms(tc)	Computes the convective terms required for the thin boundary layers equations (TBLE) and stores them in the tc.
Post.IBM.computeExtraVariables(ts, PInf, QInf)	Computes additional variables required for the IBM post-processing.
Post.IBM.loads(tb_in[, tc_in, tc2_in, ...])	Computes the viscous and pressure forces on the immersed boundaries

CHAPTER TWO

CONTENTS

Post.IBM.**extractIBMWallFields**(*tc*, *tb=None*, *coordRef='wall'*, *famZones=[ ]*, *IBCNames='IBCD_*'*, *extractIBMInfo=False*)

Projects the solution computed and stored at IBM points onto the vertices of the surface.

If tb is None, returns the cloud of IBM points. Else, the solution is projected onto the bodies, using a third-order accurate Moving Least Squares interpolation.

Returns density, pressure, utau, yplus, velocity components. (Optional: yplus at image points, pressure gradients, curvature coefficient, temperature)

Parameters

- **tc** (*[zone, list of zones, base, tree]*) – connectivity tree
- **tb** (*[zone, list of zones, base, tree]*) – surface mesh (TRI-type)
- **coordRef** (*'wall','target' or 'image'*) – reference coordinates for the cloud of IBM points (default is IBM wall points)
- **famZones** (*list of family names*) – list of IBC families to be projected
- **extractIBMInfo** (*boolean*) – if True, extracts all IBM point coordinates (wall, target and image points)

Returns

surface tree with the flow solution (density, pressure, friction velocity, yplus)

Example of use:

- Projects the solution at IBM wall points onto the vertices of the surface (pyTree):

```
# - extractionIBM a la paroi (pyTree) -
import Converter.PyTree as C
import Post.IBM as P_IBM
import Geom.PyTree as D
```

(continues on next page)

(continued from previous page)

```
import Generator.PyTree as G
import Converter.Internal as Internal
import numpy
import KCore.test as test

a = D.sphere((0,0,0),1,N=20); a=C.convertArray2Tetra(a); a = G.
↪close(a)
ts = C.newPyTree(["Base",a])
C._addState(ts, 'EquationDimension',3)
C._addState(ts, 'GoverningEquations', 'Euler')

x0 = -2; N = 41; h = 2*abs(x0)/(N-1)
z = G.cart((x0,x0,x0),(h,h,h),(N,N,N))
zname = Internal.getName(z)
zsr = Internal.createNode('IBCD_'+zname, 'ZoneSubRegion_t',␣
↪value=zname)
Internal._createChild(zsr, 'GridLocation', 'GridLocation_t', value=
↪'CellCenter')

# mimic the IBM wall pt info
a2 = D.sphere((0,0,0),1, N=30); a2 = C.convertArray2Tetra(a2); a2 =␣
↪G.close(a2)
GC = Internal.getNodeFromType(a2,"GridCoordinates_t")
FSN = Internal.getNodeFromType(a2,'FlowSolution_t')
nIBC = Internal.getZoneDim(a2)[1]
XP = numpy.zeros((nIBC),numpy.float64)
XN = Internal.getNodeFromName(GC,'CoordinateX')[1]; XP[:]=XN[:]
YP = numpy.zeros((nIBC),numpy.float64)
YN = Internal.getNodeFromName(GC,'CoordinateY')[1]; YP[:]=YN[:]
ZP = numpy.zeros((nIBC),numpy.float64)
ZN = Internal.getNodeFromName(GC,'CoordinateZ')[1]; ZP[:]=ZN[:]
DENS = numpy.ones((nIBC),numpy.float64)
DENS[:]=XP[:]*YP[:]*ZP[:]
PRESS = 101325*numpy.ones((nIBC),numpy.float64)
zsr[2].append(['CoordinateX_PW', XP, [], 'DataArray_t'])
zsr[2].append(['CoordinateY_PW', YP, [], 'DataArray_t'])
zsr[2].append(['CoordinateZ_PW', ZP, [], 'DataArray_t'])
zsr[2].append(['Pressure', PRESS, [], 'DataArray_t'])
zsr[2].append(['Density', DENS, [], 'DataArray_t'])
z[2].append(zsr)
tc = C.newPyTree(['CART']); tc[2][1][2].append(z)
z = P_IBM.extractIBMWallFields(tc, tb=ts, loc='nodes')
```

(continues on next page)

(continued from previous page)

```
C._initVars(z,'{Density0}={CoordinateX}*{CoordinateY}*{CoordinateZ}')
C.convertPyTree2File(z,"out.cgns")
```

Post.IBM.**extractShearStress**(*tb*)

Computes the shear stress on the immersed boundary surface using utau values. Exists also as in-place (_extractShearStress).

Parameters
tb (*[zone, list of zones, base, tree]*) – surface mesh (TRI-type) with density, velocity, utau variable

Returns
surface tree with the shear stress variables located at the cell centers (“ShearStressXX”, “ShearStressYY”, “ShearStressZZ”, “ShearStressXY”, “ShearStressXZ”, “ShearStressYZ”)

Example of use:

- Computes the shear stress on the immersed boundary surface using utau values. (pyTree):

Post.IBM.**extractLocalPressureGradients**(*tb*)

Computes the pressure gradients in the wall normal/tangent direction. Exists also as in-place (_extractLocalPressureGradients).

Parameters
tb (*[zone, list of zones, base, tree]*) – surface mesh (TRI-type) with pressure gradients in x, y and z directions

Returns
surface tree with the normal and tangential pressure gradient variables located at the cell centers (“gradtP” and “gradnP”)

Example of use:

Post.IBM.**extractYplusIP**(*tc*)

Computes yplus values at image points and stores them in the tc. Exists also as in-place (_extractYplusIP).

These new yplus values require yplus information located at target points as well as all IBM point coordinates (wall, target and image points)

Parameters
tc (*[zone, list of zones, base, tree]*) – connectivity tree

Returns
same as input with yplusIP field in each IBCD zone.

Example of use:

`Post.IBM.`**`extractPressureHO`**`(`*tc*, *extractDensity=False*`)`

Extrapolates the wall pressure (1st order) at the immersed boundaries and stores the solution in the tc. Exists also as in-place (_extractPressureHO).

Requires pressure gradient information in the x, y and z directions.

Parameters

- **tc** (*[zone, list of zones, base, tree]*) – connectivity tree
- **extractDensity** (*boolean*) – if True, modifies the density solution using perfect gas law and the updated pressure solution

Returns
same as input with updated pressure solution in each IBCD zone.

Example of use:

- 1st order extrapolation of the pressure at the IB (pyTree):

```
# - extractPressureHO (pyTree) -
import Converter.Internal as Internal
import Converter.PyTree as C
import Generator.PyTree as G
import Geom.PyTree as D
import KCore.test as test
import Post.IBM as P_IBM
import copy
import numpy

a = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,11,12))
a = C.node2Center(a)
for z in Internal.getZones(a):
    Internal._createChild(z, 'IBCD_2_'+z[0] , 'ZoneSubRegion_t',␣
↪value=z[0])

Nlength  = numpy.zeros((10),numpy.float64)
for z in Internal.getZones(a):
    subRegions = Internal.getNodesFromType1(z, 'ZoneSubRegion_t')
    for zsr in subRegions:
        Internal._createChild(zsr, 'ZoneRole', 'DataArray_t', value=
↪'Donor')
```

(continues on next page)

(continued from previous page)

```
        Internal._createChild(zsr, 'GridLocation', 'GridLocation_t',␣
↪value='CellCenter')

        zsr[2].append(['Pressure', Nlength+3, [], 'DataArray_t'])
        zsr[2].append(['Density' , copy.copy(Nlength)+1, [],
↪'DataArray_t'])

        zsr[2].append(['CoordinateX_PW', copy.copy(Nlength), [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateY_PW', copy.copy(Nlength), [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateZ_PW', copy.copy(Nlength), [],
↪'DataArray_t'])

        zsr[2].append(['CoordinateX_PC', copy.copy(Nlength)+14, [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateY_PC', copy.copy(Nlength)+14, [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateZ_PC', copy.copy(Nlength)+14, [],
↪'DataArray_t'])

        zsr[2].append(['CoordinateX_PI', copy.copy(Nlength)+15, [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateY_PI', copy.copy(Nlength)+15, [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateZ_PI', copy.copy(Nlength)+15, [],
↪'DataArray_t'])

        zsr[2].append(['gradxPressure', copy.copy(Nlength)+2, [],
↪'DataArray_t'])
        zsr[2].append(['gradyPressure', copy.copy(Nlength)+2, [],
↪'DataArray_t'])
        zsr[2].append(['gradzPressure', copy.copy(Nlength)+2, [],
↪'DataArray_t'])

a=P_IBM.extractPressureHO(a)
C.convertPyTree2File(a,'out.cgns')
```

Post.IBM.**extractPressureHO2**(*tc*, *extractDensity=False*)

Extrapolates the wall pressure (2nd order) at the immersed boundaries and stores the solution in the tc. Exists also as in-place (_extractPressureHO2).

Requires first and second order pressure gradient information in the x, y and z directions.

Parameters

- **tc** (*[zone, list of zones, base, tree]*) – connectivity tree
- **extractDensity** (*boolean*) – if True, modifies the density solution using perfect gas law and the updated pressure solution

Returns

same as input with updated pressure solution in each IBCD zone.

Example of use:

- 2nd order extrapolation of the pressure at the IB (pyTree):

```
# - extractPressureHO2 (pyTree) -
import Converter.Internal as Internal
import Converter.PyTree as C
import Generator.PyTree as G
import Geom.PyTree as D
import KCore.test as test
import Post.IBM as P_IBM
import copy
import numpy

a = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,11,12))
a = C.node2Center(a)
for z in Internal.getZones(a):
    Internal._createChild(z, 'IBCD_2_'+z[0] , 'ZoneSubRegion_t',␣
↪value=z[0])

Nlength  = numpy.zeros((10),numpy.float64)
for z in Internal.getZones(a):
    subRegions = Internal.getNodesFromType1(z, 'ZoneSubRegion_t')
    for zsr in subRegions:
        Internal._createChild(zsr, 'ZoneRole', 'DataArray_t', value=
↪'Donor')
        Internal._createChild(zsr, 'GridLocation', 'GridLocation_t',␣
↪value='CellCenter')

        zsr[2].append(['Pressure', Nlength+3, [], 'DataArray_t'])
        zsr[2].append(['Density' , copy.copy(Nlength)+1, [],
↪'DataArray_t'])

        zsr[2].append(['CoordinateX_PW', copy.copy(Nlength), [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateY_PW', copy.copy(Nlength), [],
↪'DataArray_t'])
```

(continues on next page)

(continued from previous page)

```
        zsr[2].append(['CoordinateZ_PW', copy.copy(Nlength), [],
↪'DataArray_t'])

        zsr[2].append(['CoordinateX_PC', copy.copy(Nlength)+14, [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateY_PC', copy.copy(Nlength)+14, [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateZ_PC', copy.copy(Nlength)+14, [],
↪'DataArray_t'])

        zsr[2].append(['CoordinateX_PI', copy.copy(Nlength)+15, [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateY_PI', copy.copy(Nlength)+15, [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateZ_PI', copy.copy(Nlength)+15, [],
↪'DataArray_t'])

        zsr[2].append(['gradxPressure', copy.copy(Nlength)+2, [],
↪'DataArray_t'])
        zsr[2].append(['gradyPressure', copy.copy(Nlength)+2, [],
↪'DataArray_t'])
        zsr[2].append(['gradzPressure', copy.copy(Nlength)+2, [],
↪'DataArray_t'])

        zsr[2].append(['gradxxPressure', copy.copy(Nlength)+3, [],
↪'DataArray_t'])
        zsr[2].append(['gradxyPressure', copy.copy(Nlength)+3, [],
↪'DataArray_t'])
        zsr[2].append(['gradxzPressure', copy.copy(Nlength)+3, [],
↪'DataArray_t'])

        zsr[2].append(['gradyxPressure', copy.copy(Nlength)+4, [],
↪'DataArray_t'])
        zsr[2].append(['gradyyPressure', copy.copy(Nlength)+4, [],
↪'DataArray_t'])
        zsr[2].append(['gradyzPressure', copy.copy(Nlength)+4, [],
↪'DataArray_t'])

        zsr[2].append(['gradzxPressure', copy.copy(Nlength)+5, [],
↪'DataArray_t'])
        zsr[2].append(['gradzyPressure', copy.copy(Nlength)+5, [],
↪'DataArray_t'])
```

(continues on next page)

(continued from previous page)

```
        zsr[2].append(['gradzzPressure', copy.copy(Nlength)+5, [],
↪'DataArray_t'])

a=P_IBM.extractPressureHO2(a)
C.convertPyTree2File(a,'out.cgns')
```

Post.IBM.**extractConvectiveTerms**(*tc*)

Computes the convective terms required for the thin boundary layers equations (TBLE) and stores them in the tc.

Requires velocity gradient information in the x, y and z directions.

Parameters

tc (*[zone, list of zones, base, tree]*) – connectivity tree

Returns

same as input with convective terms in each IBCD zone (conv1: u*(du/dx) and conv2: v*(du/dy)).

Example of use:

- Computes the convective terms (pyTree):

```
# - extractConvectiveTerms (pyTree) -
import Converter.Internal as Internal
import Converter.PyTree as C
import Generator.PyTree as G
import Geom.PyTree as D
import KCore.test as test
import Post.IBM as P_IBM
import copy
import numpy

a = G.cart((0.,0.,0.), (0.1,0.1,0.2), (10,11,12))
a = C.node2Center(a)
for z in Internal.getZones(a):
    Internal._createChild(z, 'IBCD_2_'+z[0] , 'ZoneSubRegion_t',␣
↪value=z[0])

Nlength  = numpy.zeros((10),numpy.float64)
for z in Internal.getZones(a):
    subRegions = Internal.getNodesFromType1(z, 'ZoneSubRegion_t')
    for zsr in subRegions:
        Internal._createChild(zsr, 'ZoneRole', 'DataArray_t', value=
```

(continues on next page)

(continued from previous page)

```
↪'Donor')
        Internal._createChild(zsr, 'GridLocation', 'GridLocation_t',␣
↪value='CellCenter')

        zsr[2].append(['CoordinateX_PW', copy.copy(Nlength), [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateY_PW', copy.copy(Nlength), [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateZ_PW', copy.copy(Nlength), [],
↪'DataArray_t'])

        zsr[2].append(['CoordinateX_PC', copy.copy(Nlength)+14, [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateY_PC', copy.copy(Nlength)+14, [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateZ_PC', copy.copy(Nlength)+14, [],
↪'DataArray_t'])

        zsr[2].append(['CoordinateX_PI', copy.copy(Nlength)+15, [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateY_PI', copy.copy(Nlength)+15, [],
↪'DataArray_t'])
        zsr[2].append(['CoordinateZ_PI', copy.copy(Nlength)+15, [],
↪'DataArray_t'])

        zsr[2].append(['Pressure', Nlength+3, [], 'DataArray_t'])
        zsr[2].append(['Density' , copy.copy(Nlength)+1, [],
↪'DataArray_t'])

        zsr[2].append(['gradxPressure', copy.copy(Nlength)+2, [],
↪'DataArray_t'])
        zsr[2].append(['gradyPressure', copy.copy(Nlength)+2, [],
↪'DataArray_t'])
        zsr[2].append(['gradzPressure', copy.copy(Nlength)+2, [],
↪'DataArray_t'])

        zsr[2].append(['gradxVelocityX', copy.copy(Nlength)+3, [],
↪'DataArray_t'])
        zsr[2].append(['gradyVelocityX', copy.copy(Nlength)+3, [],
↪'DataArray_t'])
        zsr[2].append(['gradzVelocityX', copy.copy(Nlength)+3, [],
↪'DataArray_t'])
```

(continues on next page)

(continued from previous page)

```
        zsr[2].append(['gradxVelocityY', copy.copy(Nlength)+4, [],
↪'DataArray_t'])
        zsr[2].append(['gradyVelocityY', copy.copy(Nlength)+4, [],
↪'DataArray_t'])
        zsr[2].append(['gradzVelocityY', copy.copy(Nlength)+4, [],
↪'DataArray_t'])

        zsr[2].append(['gradxVelocityZ', copy.copy(Nlength)+5, [],
↪'DataArray_t'])
        zsr[2].append(['gradyVelocityZ', copy.copy(Nlength)+5, [],
↪'DataArray_t'])
        zsr[2].append(['gradzVelocityZ', copy.copy(Nlength)+5, [],
↪'DataArray_t'])

        zsr[2].append(['VelocityX', copy.copy(Nlength)+6, [],
↪'DataArray_t'])
        zsr[2].append(['VelocityY', copy.copy(Nlength)+7, [],
↪'DataArray_t'])
        zsr[2].append(['VelocityZ', copy.copy(Nlength)+8, [],
↪'DataArray_t'])

a=P_IBM.extractConvectiveTerms(a)
C.convertPyTree2File(a,'out.cgns')
```

`Post.IBM.`**`computeExtraVariables`**(*tb*, *PInf*, *QInf*, *variables=['Cp', 'Cf', 'frictionX', 'frictionY', 'frictionZ', 'frictionMagnitude', 'ShearStress']*)

Computes additional variables required for the IBM post-processing. Uses density, pressure, utau, and velocity variables located at the vertices of tb.

Possible extra variables are 'Cp', 'Cf', 'frictionX', 'frictionY', 'frictionZ', 'frictionMagnitude', 'ShearStress', 'gradnP' and 'gradtP'.

Parameters

- **tb** (*[zone, list of zones, base, tree]*) – surface mesh (TRI-type) with density, pressure, utau, and velocity variables.
- **PInf** (*real*) – reference pressure to compute Cp
- **QInf** (*real*) – reference dynamic pressure
- **variables** (*list of strings*) – list of variables to be computed

Returns
surface tree with additional variables located at the cell centers

Example of use:

- Computes variables using variables density, pressure, utau, and velocity at vertices of tb (pyTree):

```
#compute shear stress for IBM
import Post.IBM as P_IBM
import Converter.PyTree as C
import Converter.Internal as Internal
import Geom.PyTree as D
import Generator.PyTree as G

a=D.sphere((0,0,0),0.5,N=30)
a = C.convertArray2Tetra(a); a = G.close(a)
C._initVars(a,'{centers:utau}={centers:CoordinateX}**2')
C._initVars(a,'{centers:VelocityX}={centers:CoordinateZ}*
↪{centers:CoordinateY}')
C._initVars(a,'{centers:VelocityY}={centers:CoordinateX}*
↪{centers:CoordinateZ}')
C._initVars(a,'{centers:VelocityZ}={centers:CoordinateX}*
↪{centers:CoordinateY}')
C._initVars(a,'{centers:Density}=1')
C._initVars(a,'{centers:Pressure}=0.71')

P_IBM._computeExtraVariables(a,PInf=0.71, QInf=0.005, variables=['Cp
↪','Cf','ShearStress'])
C.convertPyTree2File(a,"out.cgns")
```

`Post.IBM.`**`loads`**(*tb_in*, *tc_in=None*, *tc2_in=None*, *wall_out=None*, *alpha=0.*, *beta=0.*, *Sref=None*, *order=1*, *gradP=False*, *famZones=[]*)

Computes the viscous and pressure forces on the immersed boundaries (IB). If tc_in=None, tb_in must also contain the projection of the flow field solution onto the surface.

if tc and tc2 are not None, uses the pressure information at second image points.

Parameters

- **tb_in** (*[zone, list of zones, base, tree]*) – geometry tree
- **tc_in** (*[zone, list of zones, base, tree, or None]*) – connectivity tree
- **tc2_in** (*[zone, list of zones, base, tree, or None]*) – connectivity tree containing IBM information for the second image point (optional)

- **wall_out** (*string or None*) – file name for the output
- **alpha** (*float*) – angle of attack (x-y plane) (in degrees)
- **beta** (*float*) – angle of attack (x-z plane) (in degrees)
- **gradP** (*boolean*) – if True, extracts the wall pressure using pressure gradient information (see extractPressureHO() or extractPressureHO2())
- **order** (*1 or 2*) – pressure extrapolation order (when gradP is active)
- **Sref** (*float or None*) – reference surface area for calculating the aerodynamic coefficients (CD/CL). if Sref is None, Sref is computed as the surface area
- **famZones** (*list of strings or None*) – List of IBC families on which loads are computed.

Returns
surface tree with the flow solution. Lists of CD/CL per base.

Example of use:

- Computes the viscous and pressure forces on an IB (pyTree):