pyCGNS.WRA/Manual

Release 4.2.0

Marc Poinot

June 20, 2012

CONTENTS

The *CGNS/MLL* and *CGNS/ADF* wrapper. Provides the user with a *Python*-like interface to the *CGNS/MLL* and *CGNS/ADF* functions. There are two classes acting as proxies to the *MLL* and the *ADF* calls. The interface to the calls are most of the time unchanged, that is the order and the types of the arguments are more or less the same as for the C/Fortran APIs.

The file id is always the first argument in the C/Fortran API, it is replaced by the class itself which holds the required connection information.

Warning: The so-called *ADF* calls of the **v3** version of *CGNS* libraries can read/write *ADF* as well as *HDF5* files.

CHAPTER

ONE

MLL EXAMPLES

If you really want to use CGNS/MLL, for example if you have a large toolbox with old pyCGNS scripts you have to change your imports from *CGNS.WRA* to *CGNS.WRA.wrapper* for functions. There is a long example with fake data, the purpose here is to show the syntax:

```
import CGNS.WRA.wrapper
import CGNS.PAT.cgnskeywords as CGK
import numpy as NPY

db=CGNS.WRA.wrapper.pyCGNS('testfile.cgns',CGK.MODE_WRITE)
db.basewrite('Base',3,3)
db.zonewrite(1,'Zone 01',[3,5,7,2,4,6,0,0,0],CGK.Structured)
db.zonewrite(1,'Zone 02',[3,5,7,2,4,6,0,0,0],CGK.Structured)

# beware: the keywords for strings have the _s postfix like below
#         but enumerates are used without postfix
#
db.coordwrite(1,1,CGK.RealDouble,CGK.CoordinateX_s,c01[0])
db.one2onewrite(1,1,"01-02","Zone 02",(3,1,1,3,5,7),(1,1,1,1,5,7),(1,2,3))
db.bcwrite(1,1,"I low",CGK.BCTunnelInflow,CGK.PointRange,[(1,1,1),(3,2,4)])
db.bcdatasetwrite(1,1,1,"I low DATA SET",CGK.BCTunnelInflow)
db.bcdatawrite(1,1,1,1,CGK.Neumann)

# we have now nodes that can be set as children of different node types,
# then we have to set the parent node using the goto. The current parent
# is a variable for the current CGNS file.
#
db.goto(1,[])
db.statewrite(CGK.ReferenceState_s)

db.goto(1,[(CGK.ReferenceState_t,1)])
db.arraywrite("Mach",CGK.RealDouble,1,(1,),NPY.array([0.8],'d'))

db.close()
```

When a function call fails, it sets the error code and error message. You have to check it at each call if you want to make sure there is no problem:

```
import CGNS.WRA.wrapper
import CGNS.PAT.cgnskeywords as CGK

db=CGNS.WRA.wrapper.pyCGNS('testfile.cgns',CGK.MODE_WRITE)
if (db.error[0] != 0):
  print "# Error code:%d message:[%s]"%(db.error[0],db.error[1])
```

The actual data is passed to the MLL using the numpy arrays. When you write you have to create the numpy array, once it is passed to the function you can delete it:

```
w=NPY.array([[0.891,4.12],[1.0,2.2],[3.14159,3.2]],'d')
db.arraywrite("OriginLocation",CGK.RealDouble,len(w.shape),w.shape,w)
```

When you read, the MLL returns a new numpy array to you, this array now belongs to you and you can close the CGNS file and/or delete the array, there is no more relationship between the array and MLL:

```
r=db.fieldread(1,2,2,CGK.Density_s,CGK.RealDouble,[1,1,1],[2,4,6])
db.close()
print r.shape
```

CHAPTER

TWO

ADF EXAMPLES

The following example shows how to create links. A `secondfile` is open with the *ADF* interface and then we add links to the `firstfile` grid. This is a typical pattern used to share a large grid between several files:

```
import CGNS.WRA.wrapper
import CGNS.WRA._adf as ADF

firstfile="M6grid.cgns"
secondfile="M6comput.cgns"

filesol=CGNS.WRA.wrapper.pyADF(secondfile,ADF.OLD,ADF.NATIVE)
for bn in filesol.children_names(filesol.root()):
  bx=filesol.get_node_id(filesol.root(),bn)
  if (filesol.get_label(bx) == 'CGNSBase_t'):
    for zn in filesol.children_names(bx):
     zx=filesol.get_node_id(bx,zn)
     if (filesol.get_label(zx) == 'Zone_t'):
      filesol.link(zx,'GridCoordinates',
                   firstfile,'/%s/%s/GridCoordinates'%(bn,zn))
filesol.database_close()
```

The next example uses the *CGNS/MLL* calls. Again we are creating links from the `secondfile` to the `firstfile`, or more exactly from a list of `secondfiles` to the `firstfile`. The purpose of the script is to gather a set of *CGNS* files produced by a parallel computation, the `secondfile` is an empty skeletton with links to the actual files.

CHAPTER

THREE

CONTENTS

3.1 CGNS.WRA.wrapper

The CGNS/MLL library wrapper. The pyCGNS wrap classes are simple wrappers on top of the ADF and the MLL libraries.

class CGNS.WRA.wrapper.**pyADF**(*name*, *status*, *format*)

The *pyADF* class holds a connection to a *CGNS* file provides the user with a *Python*-like interface to *ADF* calls.

Each method or attribute use actually runs the ADF library functions

ADF calls to DB have the same name without the 'ADF' prefix and with a lowercase name. Argument list always try to be as close as possible to the ADF original argument list

database_close()

Closes a CGNS database. It does 'not' delete the object, so we suggest you use the Python del statement instead of this method.

database_delete(*filename*)

Not implemented at ADF library level.

class CGNS.WRA.wrapper.**pyCGNS**(*name*, *mode=-9*)

pyCGNS: wrapper to CGNS calls - pyCGNS creation opens a database - CGNS calls are partly implemented

close()

-Close the current CGNS file -(pyCGNS)

'None='*close*'()'

Close is performed by *del* if not already (explicitely) done.

goto(*baseidx*, *path=()*)

-Set the current node -Node

'node-id:D='*goto*'(base-id:I,path:((node-type:S,node-id:I),...))'

The 'goto' sets the current node to the leaf of the given path. The path itself is a list of tuples. Each tuple contains the node type a first argument, its index as second arg. Note the returned id is not (yet) trustable.

deletenode(*name*)

-Delete the given node -Node

'None='*delenode*'(name:S)'

Removes the current node (and its children).

id(*baseidx*, *path*)

-Get the ADF id of a given node -(pyCGNS)

'node-id:D='*id*'(base-id:I,path:((node-type:S,node-id:I),...))'

Uses the same syntax as 'goto', but doesn't set the current node. This id call is not trustable, it has not beeen tested and probably has a destructive effect on cgnslib global variables... See 'goto' remarks

basewrite(*name*, *cdim*, *pdim*)
-Create a new base in an existing CGNS file -Base

'base-idx:I='*basewrite*'(base-name:S,cell-dim:I,phys-dim:I)'

Args are base name, cell and physical dimensions. Returns the new id of the created base

baseread(*id*)
-Get infos about a given base -Base

'(base-idx:I,base-name:S,cell-dim:I,phys-dim:I)='*baseread*'(base-idx:I)'

Arg is base id, returns a tuple with base information.

baseid(*id*)
-Get the ADF id of a base -undocumented MLL

'base-id:D='*baseid*'(base-id:I)'

The argument is the MLL id, the return value is the ADF id, it is a double float value. Such an id cannot be obtained if the CGNS/ADF file has been open as write only.

linkread()
-Get infos about a given link -Link

'(file-name:S,target-name:S))='*linkread*'()'

Uses the current node as argument node previously set by 'goto' call.

linkwrite(*sourcenode*, *destfile*, *destnode*)
-Create a link -Link

'None='*linkwrite*'(source-name:S,file-name:S,target-name:S)'

Args are the link name, the destination file, destination node name. Returns None.

islink()
-Test if current node is a link -Link

'true-if-is-link:I='*islink*'()'

Uses current node.

descriptorread(*index*)
-Get the descriptor contents -Descriptor

'(desc-name:S,desc-text:S)='*descriptorread*'(desc-id:I)'

The current node is used.

descriptorwrite(*nodename*, *text*)
-Create or update a descriptor under the current node -Descriptor

'None='*descriptorwrite*'(desc-name:S,desc-test:S)'

Unfair-remark: We should get the descriptor id as returned argument.

nzones(*id*)
-Count zones in the base -Zone

'number-of-zones:I='*nzones*'(base-id:I)'

The zone count is a max, the zone ids are starting from 1 (one). Thus, using 'nzones' with a 'range' function should be done with 'range(1,file.nzones(base)+1)'

zoneid(*bid*, *zid*)
-Get the ADF id of a zone -undocumented MLL

'zone-id:D='*zoneid*'(base-id:I,zone-id:I)'

See 'baseid' remarks.

zoneread(*bid*, *zid*)
-Get infos about a given zone -Zone

'(base-id:I,zone-id:I,zone-name:S,size-tuple(I,...)='*zoneread*'(base-id:I,zone-id:I)'

The tuple returns a useful informatinos, including arguments ids. The dimension tuple size depends on the zone size. To get this tuple size, use the 'len' function.

zonetype(*bid*, *zid*)
-Get the type of a given zone -Zone

'zone-type:S='*zonetype*'(base-id:I,zone-id:I)'

The returned string can be used as entry key into ZoneType dictionnary, in order to get the actual integer value for the corresponding enumerate.

zonewrite(*baseidx*, *name*, *szlist*, *zonetype*)
-Create a new zone -Zone

'zone-id:I='*zonewrite*'(base-id:I,zone-name:S,size-tuple:(I,...),zone-type:S)'

See 'zonetype' remarks.

equationsetchemistryread()
-Read chemistry flags -Flow Equation Set

'equation-flags:(I,I)='*equationsetchemistryread*'()'

No Comment

equationsetread()
-Get equation set info -Flow Equation Set

'equation-dim:(I,I,I,I,I)='*equationsetread*'()'

No Comment

equationsetwrite(*d*)
-Set equation set info -Flow Equation Set

'None='*equationsetwrite*'(equation-dim:I)'

No Comment

governingread()
-Get governing equations info -Flow Equation Set

'governing-eq-type:I='*governingread*'()'

No Comment

governingwrite(*d*)
-Set governing equations info -Flow Equation Set

'None='*governingwrite*'(governing-eq-type:I)'

No Comment

diffusionread()
-Get diffusion info -Flow Equation Set

'(I,I,I,I,I)='*diffusionread*'()'

No Comment

diffusionwrite(*d*)
-Set diffusion info -Flow Equation Set

'None='*diffusionwrite*'(I,I,I,I,I)'

No Comment

modelread(*label*)
-Get model info -Flow Equation Set

'(model-name:S,model-type:I)='*modelread*'(nodel-name:S)'

No Comment

modelwrite(*label*, *mt*)
-Set model info -Flow Equation Set

'None='*modelwrite*'(model-name:S,model-type:I)'

No Comment

stateread()
-Get state info -Flow Equation Set

'state-description:S='*stateread*'()'

No Comment

statewrite(*name*)
-Set state info -Flow Equation Set

'None='*statewrite*'(state-description:S)'

No Comment

simulationtyperead(*bid*)
-Get simulation type info -Flow Equation Set

'simulation-type:I='*simulationtyperead*'(base-id:I)'

No Comment

simulationtypewrite(*bid*, *simtype*)
-Set simulation type info -Flow Equation Set

'None='*simulationtypewrite*'(base-id:I,simulation-type:I)'

No Comment

rotatingread()
-Get the rotation parameters -Rotating Coordinates

'(rate-vector:(D,...),center:(D,...))='*rotatingread*'()'

The '(D,...)' have the base physical dimension (i.e. 2 in 2D and 3 in 3d).

rotatingwrite(*rv*, *rc*)
-Set the rotation parameters -Rotating Coordinates

'None='*rotatingwrite*'(rate-vector:(D,...),center:(D,...))'

See 'rotatingread'

axisymread(*bid*)
-Get the axisymmetry parameters -Axisymmetry

'(reference-point:(D,D),axis-vector:(D,D))='*axisymread*'(base-id:I)'

Should be 2D.

axisymwrite(*bid*, *rv*, *rc*)
-Set the axisymmetry parameters -Axisymmetry

'None='*axisymwrite*'(base-id:I,reference-point:(D,D),axis-vector:(D,D))'

Should be 2D.

gravityread(*bid*)
-Get the gravity vector -Auxiliary Data

'(gravity-vector:(D,...))='*gravityread*'(base-id:I)'

Size depends on physical dimension of base.

gravitywrite(*bid*, *gv*)
-Set the gravity vector -Auxiliary Data

'None='*gravitywrite*'(base-id:I,gravity-vector:(D,...))'

Size depends on physical dimension of base.

gridlocationread()
-Get the grid location info -Grid

'grid-location:I='*gridlocationread*'()'

Under current node

gridlocationwrite(*gloc*)
-Set the grid location info -Grid

'None='*gridlocationwrite*'(grid-location:I)'

Under current node

nrigidmotions(*bid*, *zid*)
-Get number of rigid motion nodes -Rigid Grid Motion

'number-of-motion:I='*nrigidmotions*'(base-id:I,zone-id:I)'

No Comment

narbitrarymotions(*bid*, *zid*)
-Get number of arbitrary motion nodes -Arbitrary Grid Motion

'number-of-motion:I='*narbitrarymotions*'(base-id:I,zone-id:I)'

No Comment

rigidmotionread(*bid*, *zid*, *mid*)
-Get info about a rigid motion node -Rigid Grid Motion

'return-tuple='*rigidmotionread*'(base-id:I,zone-id:I,motion-id:I)'

The returned tuple contains: (name:S,RigidGridMotionType:I)

arbitrarymotionread(*bid*, *zid*, *mid*)
-Get info about an arbitrary motion node -Arbitrary Grid Motion

'return-tuple='*arbitrarymotionread*'(base-id:I,zone-id:I,motion-id:I)'

The returned tuple contains: (name:S,ArbitraryGridMotionType:I)

rigidmotionwrite(*bid*, *zid*, *name*, *type*)
-Create a new rigid motion node -Rigid Grid Motion

'rigid-motion-id:I='*rigidmotionwrite*'(base-id:I,zone-id:I,name:S,type:I)'

type:RigidGridMotionType

arbitrarymotionwrite(*bid*, *zid*, *name*, *type*)
-Create a new arbitrary motion node -Arbitrary Grid Motion

'arbitrary-motion-id:I='*arbitrarymotionwrite*'(base-id:I,zone-id:I,name:S,type:I)'

type:ArbitraryGridMotionType

nsols(*bid*, *zid*)
-Get count of solutions -Flow Solution

'number-of-solutions:I='*nsols*'(base-id:I,zone-id:I)'

No Comment

solwrite (*bid*, *zid*, *sname*, *glocation*)
-Create a new solution -Flow Solution

'sold-id:I='*solwrite*'(base-id:I,zone-id:I,sol-name:S,grid-location:S)'

The grid location is a string. The corresponding enumerate can be found using the cross dictionnary. Examples: s=db.solwrite(1,z,mySolutionName,CGNS.CellCenter) s=db.solwrite(1,z,mySolutionName,CGNS.CellCenter_[myGridLocation])

solinfo (*bid*, *zid*, *sid*)
-Get infos about a given solution -Flow Solution

'(grid-location:S,sol-name:S)='*solinfo*'(base-id:I,zone-id:I,sol-id:I)'

See 'solwrite' remarks.

solid (*bid*, *zid*, *sid*)
-Get the ADF id of a solution -undocumented MLL

'sol-id:D='*solid*'(base-id:I,zone-id:I,sol-id:I)'

See 'baseid' remarks.

ncoords (*bid*, *zid*)
-Count coordinates nodes in the zone -Coordinates

'number-of-coords:I='*ncoords*'(base-id:I,zone-id:I)'

See 'nzones' remarks.

coordread (*bid*, *zid*, *cname*, *rmode=0*)
-Read the coordinate array -Coordinates

'coord-array:A='*coordread*'(base-id:I,zone-id:I,coord-name:S,read-mode:I)'

The returned array is a 'pyArray' containing the data with the required format. Unfair-remark: a zone name is required, but all requests to nodes are done using integer ids. The read mode is default to 0, that is a C-like read (i,j,k). The mode=1 is fortran like read (k,j,i). Please, take care of the dimensions in that case, see the 'zoneread' remarks.

coordwrite (*bid*, *zid*, *dtype*, *cname*, *darray*)
-Create a coordinate array node -Coordinates

'coord-id:I='*coordwrite*'(base-id:I,zone-id:I,data-type:S,node-name:S,data-array:A)'

data-type:DataType

coordinfo (*bid*, *zid*, *cid*)
-Get infos about a given coordinate -Coordinates

'(data-type:S,node-name:S)='*coordinfo*'(base-id:I,zone-id:I,coord-id:I)'

See 'coordwrite' remarks.

coordid (*bid*, *zid*, *cid*)
-Get the ADF id of a coordinate -undocumented MLL

'coord-id:D='*coordid*'(base-id:I,zone-id:I,coord-id:I)'

See 'baseid' remarks.

nsections (*bid*, *zid*)
-Get lower range index -Element Connectivity

'section-index:I='*nsections*'(base-id:I,zone-id:I)'

The lower range is (imin, jmin, kmin)

sectionread (*bid*, *zid*, *sid*)
-Get infos about a given section -Element Connectivity

'return-tuple='*sectionread*'(base-id:I,zone-id:I,section-id:I)'

Returns a tuple containing 'name:S' of the section, its 'type:I' 'start:I' and 'end:I', 'last-bnd-index:I', 'parent-flag:I'.

sectionwrite (*bid*, *zid*, *name*, *type*, *start*, *end*, *nb*, *ar*)
-Write a section -Element Connectivity

'section-id='*sectionwrite*'(base-id:I,zone-id:I,section-name:S,args...)'

The trailing args are the 'type:I' of the section elements the 'start:I' and 'end:I' indices, 'last-bnd-index:I' index and at last the 'elements:A' array itself (of type 'type).

elementsread (*bid*, *zid*, *sid*)
-Get elements of a section -Element Connectivity

'(elements:A,parents:A)='*elementsread*'(base-id:I,zone-id:I,section-id:I)'

Returns two arrays of I

parentdatawrite (*bid*, *zid*, *sid*, *ar*)
-Write the parent data in a section -Element Connectivity

'None='*parentdatawrite*'(base-id:I,zone-id:I,section-id:I,parent-data:A)'

No return

npe (*t*)
-Get the number of nodes for an element type -Element Connectivity

'number:I='*npe*'(element-type:I)'

element-type is an enumerate.

elementdatasize (*bid*, *zid*, *sid*)
-Get the number of elements for this section -Element Connectivity

'number:I='*elementdatasize*'(base-id:I,zone-id:I,section-id:I)'

See 'section-write'

nfields (*bid*, *zid*, *sid*)
-Count fields in the solution -Flow Solution

'number-of-fields:I='*nfields*'(base-id:I,zone-id:I,sol-id:I)'

See 'nzones' remarks.

fieldread (*bid*, *zid*, *sid*, *fname*, *dtype*, *imin*, *imax*)
-Get the data array of a given solution field -Flow Solution

'data-array:A='*fieldread*'(base-id:I,zone-id:I,sol-id:I,field-name:S,args...)'

The trailing args are 'data-type:S' and tuples of indices: 'i-min:(I,I,I)' 'i-max:(I,I,I)'. These 'imin' and 'imax' tuples are forced to 3D, but only relevant values are used. Other values can be set to zero. Unfair-remark: field name is required.

fieldwrite (*bid*, *zid*, *sid*, *dtype*, *fname*, *darray*)
-Create a data array for a solution field -Flow Solution

'field-id:I='*fieldwrite*'(base-id:I,zone-id:I,sol-id:I,args...)'

The trailing args are 'data-type:S', 'field-name:S' and the array of data 'data-array:A'. See also 'coordwrite' remarks.

fieldinfo (*bid*, *zid*, *sid*, *fid*)
-Get infos about a given solution field -Flow Solution

'(data-type:S,field-name:S)='*fieldinfo*'(base-id:I,zone-id:I,sol-id:I,field-id:I)'

See 'coordwrite' remarks.

fieldid(*bid*, *zid*, *sid*, *fid*)
-Get the ADF id of a solution field -undocumented MLL

'field-id:D='*fieldid*'(base-id:I,zone-id:I,sol-id:I,field-id:I)'

See 'baseid' remarks.

convergencewrite(*nit*, *name*)
-Craete of update a convergence node -Convergence

'None='*convergencewrite*'(number-of-iteration:I,node-name:S)'

Uses the current node. Should return the node id.

convergenceread()
-Get the convergence under current node -Convergence

'(number-of-iteration:I,node-name:S)='*convergenceread*'()'

No args, current node is used.

nholes(*bid*, *zid*)
-Get the count of overset holes -Overset Holes

'number-of-holes:I='*nholes*'(base-id:I,zone-id:I)'

Returns the number of overset holes in the current zone

holeinfo(*bid*, *zid*, *hid*)
-Get info from a given overset hole node -Overset Holes

'return-tuple='*holeinfo*'(base-id:I,zone-id:I,hole-id:I)'

Returns a tuple containing 'name:S' of the overset hole, 'grid-location:I' of the returned set(s) of points, the 'point-set-type:I', the 'number-of-point-sets:I' and the /number-of-points-per-point-set:I'. Should be called before *holeread* in order to have array dimensions before allocation.

holewrite(*bid*, *zid*, *hname*, *gloc*, *psett*, *ar*)
-Create a new overset hole node -Overset Holes

'hole-id:I='*holewrite*'(base-id:I,zone-id:I,hole-name:S,g-location:S,point-array:A)'

The grid location is a string. The corresponding enumerate can be found using the cross dictionnary.

holeread(*bid*, *zid*, *hid*)
-Get info from a given overset hole node -Overset Holes

'point-array:A'='*holeread*'(base-id:I,zone-id:I,hole-id:I)'

Gets the array containing the points. Dimensions depens on the PointSetType (see *holeinfo*).

nbc(*bid*, *zid*)
-Get the count of BC -Boundary Condition

'number-of-bc:I='*nbc*'(base-id:I,zone-id:I)'

No comment

bcinfo(*bid*, *zid*, *bcid*)
-Get info from a given BC -Boundary Condition

'return-tuple='*bcinfo*'(base-id:I,zone-id:I,bc-id:I)'

The result tuple has the following members, in that order. The 'name:S' of the node, its 'bc-type:I' and its 'point-set-type:I'. The 'number-of-points:I', the 'normal-index:(I,I,I)', the 'data-type:I' for the normals, the 'normal-flag:I' and the 'number-of-bc-data-set:I'.

bcread(*bid*, *zid*, *bcid*)
-Read point and normal lists from a given BC -Boundary Condition

'(point-list:A,normal-list:A)='*bcread*'(base-id:I,zone-id:I,bc-id:I)'

Comment

bcwrite(*bid*, *zid*, *bcname*, *bctype*, *pttype*, *ptlist*)
-Create a new BC -Boundary Condition

'bc-id:I='*bcwrite*'(base-id:I,zone-id:I,args...)'

The trailing arguments are the following, in that order. The 'bc-name:S', 'bc-type:I', 'bc-point-set-type:I', the 'point-set-list:((I,I,I),...)'. The number of points in the point set list is deduced from the length of the point list, except if the point set type is PointRange. In that case, the number of points is forced to 2.

bcid(*bid*, *zid*, *bcid*)
-Get the BC ADF id -undocumented MLL

'bc-id:D='*bcid*'(base-id:I,zone-id:I,bc-id:I)'

Used for ADF functions

bcnormalwrite(*bid*, *zid*, *bcid*, *nindex*, *nflags*, *dt*, *nlist*)
-Write the normals of a given BC -Boundary Condition

'None='*bcnormalwrite*'(base-id:I,zone-id:I,bc-id:I,args...)'

The trailing args are the following, in this order. The 'normal-index:(I,I,I)', 'normal-flag:I', 'data-type:I', 'normal-list:((D,D,D),...). Caution: normal-flag is forced to FALSE, normal-list is not taken into account.

bcdatasetwrite(*bid*, *zid*, *bcid*, *dsname*, *dstype*)
-Write the dataset set of a given BC -Boundary Condition

'dset-id:I='*bcdatasetwrite*'(base-id:I,zone-id:I,bc-id:I,dset-name:S,dset-type:I)'

dset-type:BCType

bcdatasetread(*bid*, *zid*, *bcid*, *dsid*)
-Read the dataset set of a given BC -Boundary Condition

'return-tuple='*bcdatasetread*'(base-id:I,zone-id:I,bc-id:I,dset-id:I)'

The return tuple is the following. (dset-name:S,bc-data-type:I,dir-flag:I,neu-flag:I)

bcdatawrite(*bid*, *zid*, *bcid*, *dsid*, *bctype*)
-Write the data in a BC dataset -Boundary Condition

'None='*bcdatawrite*'(base-id:I,zone-id:I,bc-id:I,dset-id:I,bc-data-type:I)'

bc-data-type:BCDataType

ndiscrete(*bid*, *zid*)
-Get count of discrete node -Discrete Data

'number-of-discrete:I='*ndiscrete*'(base-id:I,zone-id:I)'

No Comment

discretewrite(*bid*, *zid*, *name*)
-Create a new discrete data node -Discrete Data

'dics-id:I='*discretewrite*'(base-id:I,zone-id:I,disc-name:S)'

No Comment

discreteread(*bid*, *zid*, *did*)
-Get the name of discrete node -Discrete Data

'disc-name:S='*discreteread*'(base-id:I,zone-id:I,disc-id:I)'

No Comment

ngrids(*bid*, *zid*)
-Count the number of grids -Grid

'number-of-grids:I='*ngrids*'(base-id:I,zone-id:I)'

Comment

gridwrite(*bid*, *zid*, *name*)
-Create a new grid node -Grid

'grid-id:I='*function*'(base-id:I,zone-id:I,grid-name:S)'

Comment

gridread(*bid*, *zid*, *did*)
-Get the grid name -Grid

'grid-name:S='*gridread*'(base-id:I,zone-id:I,grid-id:I)'

Comment

nintegrals()
-Get count of integral data nodes -Integral Data

'number-of-integral='*nintegrals*'()'

Counts under current node

integralwrite(*name*)
-Create a new integral node -Integral Data

'integral-id:I='*integralwrite*'(integral-name:S)'

Under current node.

integralread(*id*)
-Get the name of integral node -Integral Data

'integral-name:S='*integralread*'(integral-id:I)'

Under current node.

nuserdata()
-Count number of user data -User Data

'number-of-userdata:I='*nuserdata*'()'

Under current node.

userdatawrite(*name*)
-Create a new userdata node -User Data

'userdata-id:I='*userdatawrite*'(userdata-name:S)'

Under current node, which should be set by a goto call. Actually returns the new UserData node id. It is supposed that the goto call is made by pyCGNS.

userdataread(*id*)
-Get name of the given user data id -User Data

'(userdata-id:i,userdata-name:S)='*userdataread*'(userdata-id:I)'

Under current node.

unitswrite(*mass*, *leng*, *time*, *temp*, *angl*)
-Create or update a units set under current node -Units and Dimensionals

'None='*unitswrite*'(mass-u:S,length-u:S,time-u:S,temp-u:S,angle-u:S)'

See remarks about the constants dictionnary, one can either use the defined strings, variables or their enumerates. *should be much more documented/checked here*

unitsread()
-Get the units under current node -Units and Dimensionals

'(mass-u:S,length-u:S,time-u:S,temp-u:S,angle-u:S)='*unitsread*'()'

See 'unitswrite' remarks.

dataclasswrite(*dt*)
-Create or update the dataclass under current node -Units and Dimensionals

'None='*dataclasswrite*'(data-class:I)'

The 'data-class' is a 'DataClass' enumerate.

dataclassread()
-Get the dataclass under current node -Units and Dimensionals

'data-class:I='*dataclassread*'()'

See 'dataclasswrite' remarks.

exponentswrite(*dt*, *v*)
-Create or update the exponents -Units and Dimensionals

'None='*exponentswrite*'(data-type:I,(D,D,D,D,D))'

Exponents values are: Mass, Length, Time, Temperature, Angle.

exponentsread()
-Get the exponents values -Units and Dimensionals

'(D,D,D,D,D)='*exponentsread*'()'

See 'exponentswrite' remarks.

exponentsinfo()
-Get the exponents datatype -Units and Dimensionals

'datatype:I='*exponentsinfo*'()'

Python only handles double. Beware at write time, you can have double/single.

conversionwrite(*dt*, *v*)
-Create or update the conversion factors -Units and Dimensionals

'None='*conversionwrite*'(data-type:I,(D,D))'

Conversion values are: scale, offset

conversionread()
-Get the conversion values -Units and Dimensionals

'(D,D)='*conversionread*'()'

See 'conversionwrite' remarks.

conversioninfo()
-Get the conversion datatype -Units and Dimensionals

'datatype:I='*conversioninfo*'()'

Python only handles double. Beware at write time, you can have double/single.

ordinalwrite(*o*)
-Create or update an ordinal node under current node -Ordinal

'None='*ordinalwrite*'(ordinal:I)'

Comment

ordinalread()
-Get the ordinal under current node -Ordinal

'ordinal:I='*ordinalread*'()'

Comment

bcwallfunctionread(*bid*, *zid*, *bcid*)
-Get the type of BC wallfunction -Boundary Condition

'WallFunctionType:I='*bcwallfunctionread*'(base-id:I,zone-id:I,bc-id:I)'

Comment

bcwallfunctionwrite(*bid*, *zid*, *bcid*, *type*)
-Set the type of BC wallfunction -Boundary Condition

'None='*bcwallfunctionwrite*'(base-id:I,zone-id:I,bc-id:I,type:I)'

Type is WallFunctionType

bcarearead(*bid*, *zid*, *bcid*)
-Get the area parameters -Boundary Condition

'(region:S,type:I,surf:D)='*bcarearead*'(base-id:I,zone-id:I,bc-id:I)'

Type is AreaType

bcareawrite(*bid*, *zid*, *bcid*, *type*, *surf*, *region*)
-Set the area parameters -Boundary Condition

'None='*bcareawrite*'(base-id:I,zone-id:I,bc-id:I,type:I,surf:D,region:S)'

Type is AreaType

rindwrite(*rind*)
-Create of update the rind indices under current node -Grid

'None='*rindwrite*'((imin:I,imax:I,jmin:I,jmax:I,kmin,kmax:I))'

Uses the current node. Tuple depends on dimensions, J or K could be unused in the case of 1D, 2D. Always give 6 integers, set them to zero if you are not 3D

rindread()
-Get the rind indices under current node -Grid

'(imin:I,imax:I,jmin:I,jmax:I,kmin,kmax:I)='*rindread*'()'

See 'rindwrite' comment.

nconns(*bid*, *zid*)
-Get number of generalized connectivities -Grid Connectivity

'number:I='*nconns*'(base-id:I,zone-id:I)'

No comment.

conninfo(*bid*, *zid*, *cid*)
-Get information about generalized connect node -Grid Connectivity

'return-tuple='*conninfo*'(base-id:I,zone-id:I,connect-id:I)'

The return tuple contains: 'connect-name:S', 'gridlocation:I', 'gridconnectivity:I', 'pointset-type:I', 'number-of-points:I', 'donor-name:S', 'donor-zone-type:I', donor-point-set-type:I', 'donor-data-type:I' and 'donor-number-of-points:I'

connread(*bid*, *zid*, *cid*)
-Get generalized connectivity points -Grid Connectivity

'return-tuple='*connread*'(base-id:I,zone-id:I,connect-id:I)'

The tuple contains two arrays of integers 'target-interface-points:A' and 'donor-interface-oints:A'.

connwrite(*bid*, *zid*, *name*, *gl*, *gt*, *pst*, *npt*, *pt*, *dname*, *dzt*, *dpst*, *ddt*, *dnpt*, *dpt*)
-Create a generalized connectivity node -Grid Connectivity

'connect-id:I='*connwrite*'(args)'

The arguments are defining (in this order) the 'base-id:I','zone-id:I' of the new node, its 'name:S', the 'gridlocation:I', 'gridconnectivitytype:I' and 'point-set-type:I' of the current (target) node interface. The 'number-of-points:I' and 'interface-points:A' which is an array of integers. Then the 'donor-name:S', its 'zonetype:I', 'point-set-type:I' and 'data-type:I', the 'number-of-donor-points:I' and the actual array of points 'donor-points:A'.

Note the 'DataType' is force to 'Integer'.

connaverageread(*bid*, *zid*, *cid*)
-Get special connect properties -Special Grid Connectivity

'averate-type:I='*connaverageread*'(base-id:I,zone-id:I,connect-id:I)'

The type is 'AverageInterfaceType'.

connaveragewrite(*bid*, *zid*, *cid*, *at*)
-Set special connect properties -Special Grid Connectivity

'None='*connaveragewrite*'(base-id:I,zone-id:I,connect-id:I,averate-type:I)'

The type is 'AverageInterfaceType'.

connperiodicread(*bid*, *zid*, *cid*)
-Get special connect properties -Special Grid Connectivity

'return-tuple='*connperiodicread*'(base-id:I,zone-id:I,connect-id:I)'

The size of arrays is the base physical dimension. The reurn-tuple is '(rot-center:A,rot-angle:A,translation:A)'.

connperiodicwrite(*bid*, *zid*, *cid*, *rc*, *ra*, *tt*)
-Set special connect properties -Special Grid Connectivity

'None='*connperiodicwrite*'(base-id:I,zone-id:I,connect-id:I,args...)'

the trailing args are 'rot-center:A', 'rot-angle:A' , 'translation:A'. The size of these arrays is the base physical dimension.

none2oneglobal(*bid*)
-Count the one2one nodes for the whole base -Grid Connectivity

'number-of-one2one:I='*none2oneglobal*'(base-id:I)'

Comment

none2one(*bid*, *zid*)
-Count the one2one nodes -Grid Connectivity

'number-of-one2one:I='*none2one*'(base-id:I,zone-id:I)'

Comment

one2oneid(*bid*, *zid*, *id*)
-Get the ADF id of a one2one node -undocumented MLL

'one2one-id:D='*one2oneid*'(base-id:I,zone-id:I,one2one-id:I)'

Comment

one2oneread(*bid*, *zid*, *id*)
-Get the one2one node informations -Grid Connectivity

'return-tuple='*one2oneread*'(base-id:I,zone-id:I,one2one-id:I)'

The return tuple has the following members, in that order. The 'name:S' of the node, the 'donor-name:S', the 'range' tuple and the 'donor-range' tuple which are both six-integer tuples. Then the 'transform' tuple is a three-integer tuple.

one2onereadglobal(*bid*)
-Get the one2one node informations for whole base -Grid Connectivity

'return-list='*one2onereadglobal*'(base-id:I)'

The return is a list of tuples, each tuple tuple has the following members, in that order. The 'name:S' of the node, the 'zone:S' name for which the conectivity information is related, the 'donor-name:S', the 'range' tuple and the 'donor-range' tuple which are both six-integer tuples. Then the 'transform' tuple is a three-integer tuple.

one2onewrite(*bid*, *zid*, *name*, *donor*, *wrange*, *wdonorrange*, *transform*)
-Create a 1to1 connectivity node -Grid Connectivity

'one2one-id:I='*one2onewrite*'(base-id:I,zone-id:I,args...)'

The trailing arguments are the following, in that order. The 'name:S' of the node, the 'donor-name:S', the 'range' tuple and the 'donor-range' tuple which are both six-integer tuples. Then the 'transform' tuple is a three-integer tuple.

nfamilies(*bid*)
-Count families in the base -Families

'number-of-families:I='*nfamilies*'(base-id:I)'

No Comment

familyread(*bid*, *fid*)
-Get info about a given family -Families

'(fam-name:S,number-of-fam-bc:I,number-of-geo:I)='*familyread*'(base-id:I,fam-id:I)'

No Comment

familywrite(*bid*, *name*)
-Create a new family node -Families

'fam-id:I='*familywrite*'(base-id:I,fam-name:S)'

No Comment

familynameread()
-Get name of current node family -Families

'fam-name:S='*familynameread*'()'

No Comment

familynamewrite(*name*)
-Creates name of current node family -Families

'None='*familynamewrite*'(fam-name:S)'

No Comment

familybocoread(*bid*, *fid*, *bcid*)
-Get name of current node family for a given BC -Families

'(bc-name:S,bc-type:I)='*familybocoread*'(base-id:I,fam-id:I,bc-id:I)'

No Comment

familybocowrite(*bid*, *fid*, *name*, *btype*)
-Create name of current node family for a given BC -Families

'bc-id:I='*familyboconamewrite*'(base-id:I,fam-id:I,bc-name:S,bc-type:I)'

No Comment

georead(*bid*, *fid*, *gid*)
-Get geometry info -Families

'(geo-name:S,file:S,CAD:S,parts:I)='*georead*'(base-id:I,fam-id:I,geo-id:I)'

No Comment

geowrite(*bid*, *fid*, *gname*, *fname*, *cadname*)
-Creates geometry info -Families

'geo-id:I='*geowrite*'(base-id:I,fam-id:I,geo-name:S,file:S,CAD:S)'

No Comment

partread(*bid*, *fid*, *gid*, *pid*)
-Get part info -Families

'part-name:S='*partread*'(base-id:I,fam-id:I,geo-id:I,part-id:I)'

No Comment

partwrite(*bid*, *fid*, *gid*, *pname*)
-Create part info -Families

'part-id:I='*partwrite*'(base-id:I,fam-id:I,geo-id:I,part-name:S)'

No Comment

biterread(*bid*)
-Get the name of the iterative data base -Iterative Data

'(name:S,number-of-it:I)='*biterread*'(base-id:I)'

No Comment

biterwrite(*bid*, *name*, *it*)
-Create base iterative data -Iterative Data

'None='*biterwrite*'(base-id:I,name:S,iteration:I)'

No Comment

ziterread(*bid*, *zid*)
-Get the name of the iterative data zone -Iterative Data

'name:S='*ziterread*'(base-id:I,zone-id:I)'

No Comment

ziterwrite(*bid*, *zid*, *name*)
-Create zone iterative data -Iterative Data

'None='*ziterwrite*'(base-id:I,zone-id:I,name:S)'

No Comment

narrays()
-Count arrays under the current node -Data Array

'number-of-arrays:I='*narrays*'()'

Use 'goto' to set the current node.

arrayread(*aid*)
-Get the array data -Data Array

'data-array:A='*arrayread*'(array-id:I)'

The array id its index under the current node. See 'arrayinfo' remarks.

arraywrite(*name*, *dtype*, *dim*, *ddim*, *darray*)
-Create or update a new array -Data Array

'array-id:I='*arraywrite*'(array-name:S,d-type:S,d-dim:I,d-vector:(I,...),d-array:A)'

All *type*, *dim vector* are refering to the *array* of data itself. See also 'arrayinfo' remarks.

arrayinfo(*aid*)
-Get infos about a given array -Data Array

'(array-name:S,data-type:S,data-dim:I,data-vector:(I,...))='*arrayinfo*'(array-id:I)'

The current node is the parent node of the requested array. The data-type enumerate can be found using the cross dictionnary. There is redondancy of 'data-dim' and 'data-vector', first can be deduced from the second.

lasterror()
-Get the current error -(pyCGNS)

'(error-code:I,error-message:S)='*lasterror*'()'

The 'error' attribute has the same contents.

bases()
-Count number of bases -Base

'nbases:I='*bases*'()'

The 'nbases' attribute has the same contents.

descriptors()
-Count number of descriptors -Descriptor

'ndescriptor:I='*descriptors*'()'

The 'ndescriptor' attribute has the same contents.

libversion()
-Get library version -(pyCGNS)

'version:D='*libversion*'()'

The 'version' attribute has the same contents.

name()
-Get file name -(pyCGNS)

'filename:S='*name*'()'

The 'name' attribute has the same contents.

adfroot()
-Get ADF root id -(pyCGNS)

'root-id:D='*adfroot*'()'

The 'root' attribute has the same contents.

3.2 Handling ADF and HDF5 file formats

Starting with the **v3** of *CGNS/MLL*, two file formats are now managed by the library: *ADF* and *HDF5*. If you build the *CGNS/MLL* library (that is the `libcgns.a`) with the default parameters, you will obtain a library with capabilities to handle these two file formats. The library tries to detect the format by scanning the actual file.

The shell variables `HDF5_LINK_PATH` or `ADF_LINK_PATH` are used to find the linked-to files. The syntax of these variables is the same as the usual `PATH` or `LD_LIBRAR_PATH` of your Unix shell.

CHAPTER

FOUR

WRA INDEX

- *genindex*

PYTHON MODULE INDEX