SIDS-to-Python (CGNS/Python)

Release 3.1.2

The CGNS Steering Committee

CONTENTS

This document is part of the CGNS documentation. The SIDS-to-Python document describes the CGNS/SIDS mapping with the Python programming language.

This is version 0.1 of the mapping, approved by the CGNS Steering Committee on March the 2nd 2011.

More information can be found in http://www.cgns.org

CHAPTER

ONE

CGNS/PYTHON TREE

The CGNS/Python mapping defines a **tree** structure composed of **nodes** implemented for the *Python* programming langage. A special **links** structure is also defined for a correct mapping of the management of files on the disk. The mapping presented here is *NOT* a library [1], it is the lowest possible correspondance between a CGNS/SIDS structure and a Python representation. This specification is public and could be used as the basis for Python based CGNS application interoperability. *Python* is an interpreted langage and it has a textual representation of its objects, this representation can be used for CGNS/Python trees as well.

1.1 Commitment with CGNS standard

The mapping of the SIDS into a CGNS/Python structure uses the node as atomic structure. Comparing to CGNS/ADF or CGNS/HDF5, the contents of a node is unchanged in CGNS/Python. The way we represent data is different but all nodes attributes found in the section 6 of the *SIDS-to-ADF File Mapping Manual* [2] are applicable to the CGNS/Python mapping.

The **data type** mapping is changed compared to CGNS/ADF or CGNS/HDF5, the actual representation of basic types such as integers, floats and strings are closely mapped to the Python data types. See the table *Data types*.

Other elements of the node description are like the CGNS/ADF or CGNS/HDF5 mappings, in particular the **dimensions** and the order of these dimensions. The CGNS/SIDS section 3.1 states that the dimensions order should be the so-called *Fortran indexing convention* which states the column index is the first. The CGNS/Python nodes should respect this requirements.

> **Warning:** The Python arrays can be defined with either a *C* or a *Fortran flag*, this *flag* is used to set or to find the order used for the internal storage of an array. It has **no effect** on the dimensions of a *numpy* array, but on its internal memory layout. It's up to the user to manage this flag and its impact on the use of an array, in particular for the read/write on the disks through the C API.

For example, section 6.1.2.2 describes the `DimensionalUnits_t` node with dimensions values `(32,5)`. This should be understood as *Fortran* order values, and thus `(32,5)` should be found as this in the *shape* of the *numpy* array [3] whichever status the *Fortran* flag set has.

A *numpy* array with the *C* flag set should also have a shape of `(32,5)`, again, the internal representation of this *C* array has to be taken into account during read/write operations.

See the *C API* and *Examples and Tips* sections about this requirement and its impact on *numpy* array use.

[1] The Open Source *pyCGNS* Python module defines services on the top of this CGNS/Python mapping.

[2] The *SIDS-to-ADF* [CG2] or *SIDS-toHDF5* [CG3] documents of *cgns.org* have the detailled description of each node of the standard.

[3] We show in the textual representation section that this dimension ordering could lead to quite complicated Python code, but our choice was to take the implementation from the 'Fortran' world, which is the basis of the CFD world. It would be up to the user to write his own application layer for a better Python interface.

1.2 The node structure

The structure of a CGNS data set is held in a so-called **CGNS/Python tree**. The tree is composed of nodes, each node may have children which are nodes too. The node structure is a python sequence (i.e. list or tuple), composed of four entries: the name, the value, the list of children and the type.

Attribute	*type*
Name	string
Value	*numpy* array
Children	list of CGNS/Python nodes
Type	string

The CGNS/Python mapping requires that:

> The *name* is a Python string, it should not be empty. The name should not have more than 32 chars and should not have `/` in it. The names `.` (a single dot) And `..` (dot dot) are forbidden (but names with dot and something else are allowed, for example `Zone.001` is ok).
>
> The representation of *values* uses the numpy library array. It makes it possible to share an already existing memory zone with the Python object. The numpy mapping of the values is detailled hereafter. An empty value should be represented as None, any other value is forbidden.
>
> The *children list* can be a list or a tuple. The use of a list is strongly recommended but not mandatory. A read-only tree can be declared as a tuple. It is the responsibility of the applications to parse sequences wether these are lists or tuples. A node without child has the empty list `[]` as children list.
>
> The *type* is the Python string representation of the CGNS/SIDS type [4] (i.e. it is the same for CGNS/ADF or CGNS/HDF5). A type string cannot be empty.

We have now a typical CGNS/Python node, which can be represented with the pattern [5]:

```
node = [ <name:string>, <value:numpy.array>, [ <child:node>* ], <cgns-type:string> ]
```

We use there the textual representation of a Python object. All the Python types used in this CGNS/Python mapping have a full textual representation. This is detailled in the next section.

The order of the values is significant, for example `node[0]` should always be the name of the node (Python has an index ordering starting with zero)

We see now that a CGNS/Python tree is a node. This node has children which have children and so on... Any node can be held as a subpart of a complete tree, we say each node is a *sub-tree*. Our CGNS/Python tree has a *root* node which is its first node. There is no clear definition of a *root* node in the CGNS/SIDS or in the SIDS mappings.

In the case of a *CGNSBase_t* level node, the CGNS/ADF or CGNS/HDF5 defines a sound node which can be mapped to CGNS/Python. However, the CGNS/SIDS states that several bases can be found in a CGNS tree. The father node of a base would have the pattern:

```
root = [ <CGNSLibraryVersion:node>, <CGNSBase:node>* ]
```

Which is not consistent with a *normal* node. We want to remove this exception, we define a CGNS/Python tree root, or first node, as a list with a compliant CGNS/Python node. which is not the *node* pattern. Then the applications have to have a specific way to manage this first node. This lack of *root* node is not that important when you use the CGNS/MLL because the function are hidding the actual node implementation. With CGNS/Python, the user can manage the nodes as true Python objects, and we have to provide him with a sound interface, or at least as sound as possible. For this consistency reason, the CGNS/Python mapping defines a new type for the *root* node, see the *CGNSTree_t type* section.

[4] The CGNS/SIDS type (see [CG1]) is the type of the node, *NOT* the type of the data contained into the node.

[5] The syntax is: `<A:T>` with `A` attribute name and `T` attribute type. The types are detailled in another section. The `<A:T>*` means zero or more `<A:T>` separated by `,` if more than one.

1.3 Textual representation

It is possible to declare a CGNS/Python node as a textual representation. There is a exemple of a zone connectivity sub-tree with the CGNS/Python in textual mode, a simple `PointRange` node with two 3D indices:

```
pr=['PointRange',
    numpy.array([[1,25],[1,9],[1,1]],dtype=numpy.int32,order='Fortran'),
    [],
    'IndexRange_t']
```

The `PointRange` node has no child, the children list is an empty list. The values of the array are initialized with a list, the order of the elements in the list matches the *Fortran* indexing: in that example the first point indices are `[1,1,1]` and the second point indicies are `[25,9,1]`.

The evaluation of this string by the Python interpreter creates a CGNS/Python compliant node as a Python list. Please note the types of this `pr` node, there are only native Python types (list, string, integer) and *numpy* types or enumerates. You have to have a variable to hold the node or the CGNS sub-tree, if you have no reference to the actually created Python objects these will be unreachable and thus garbaged.

The textual representation can be *import*-ed as any Python textual file, with all possible Python use you can imagine.

Warning: The Python lists are objects. When you refer to a list you do not copy this list unless you ask for such a copy. This is important because if you modify an existing list you modify an object that could be used by others. In the CGNS/Python mapping the children of a node is a list of nodes. If you refer to such a list without a copy, any modification of this child list will impact nodes using this list. This is detailled in the section *Examples and Tips* .

1.4 Numpy array mapping

A CGNS/Python node value is a *numpy* array, this python object contains the **number of dimensions**, the **dimensions**, the **data type** and the actual **data** array. Then this implicit information is not a part of the *node* structure. As we really want to have the most generic node as possible, we require that even single dimension values should be stored as *numpy* array. A single integer, float or a single string should be embedded into a *numpy* array.

As we mentionned before, an empty value has to be represented by `None` which is a native Python value, not a *numpy* value:

```
gc=['Grid#002',None,[cx,cy,cz],'GridCoordinates_t']
```

Here `cx`, `cy`, `cz`, are nodes, not arrays.

The *numpy* end-user interface makes it possible to define some of these required data as deduction of required parameters. The number of dimensions is the size of the so-called shape. The dimensions can be forced for empty values or can be deduced from the data itself:

```
a=numpy.array([1.4])
b=numpy.ones((5,7,3),'i')
```

The first declaration has dimension 1, number of dims 1, data type `float64`, all deduced from the data declaration, the second has dimensions `(5,3,7)`, number of dimensions 3, data type set as `int32`.

A *numpy* array can be declared as *C order* or *Fortran order*. There is no requirements in this mapping wether the internal layout of the memory should be *C* or *Fortran*. However, an array should have a shape with the same order of dimensions as described in the *SIDS-to-ADF File Mapping Manual* ([CG2]).

Warning: If you use the Python C API, it is the responsability to the application to check the *numpy* ordering flag and to manage the arrays with respect to memory layout. See the *C API* section.

The way to get the node data information regarding the [CG2] datatypes and dimensions requirements is to access to the *numpy* object attributes:

```
pr=numpy.array([[1,2,3],[4,5,6]])

dims=pr.shape
ndims=len(pr.shape)
datatype=pr.dtype
fortranorder=numpy.isfortran(pr)
corder=not numpy.isfortran(pr)
```

1.5 Data types

A value is a numpy array, the contents of an array is homogeneous and has a data type. The data types of your CGNS/Python arrays depends on the data type as defined in [CG2].

The type of the data can be set at the creation time, the *numpy* type is associated to the *ADF* type required by the CGNS/SIDS. A bad data type, even if it silently looks like the result you want, would lead to an non-compliant CGNS tree. The required mapping for the end-user interface uses the types :

ADF type	*Numpy type(s)*	*Remarks*
I4	*'i' int32*	(1)
I8	*'l' int64*	(2)
R4	*'f' float32*	(3)
R8	*'d' float64*	(4)
C1	*'c' '|S1'*	(5)

All other *ADF* or *numpy* types are ignored. The string type is a bit special, see the remark (5) about the strings used in *numpy* arrays.

Remarks:

1. The 32bits precision has to be forced, the default integer size in python the `int64` data type. To create an *I4* array, you can use:

   ```
   numpy.array([1,2,3],'i',order='Fortran')
   ```

2. The 64bits precision is the default integer in python. To create an *I8* array, you can use:

   ```
   numpy.array([1,2,3],order='Fortran')
   ```

3. The 32bits precision has to be forced, the default float size in python is `float64`. To create an *R4* array, you can use:

   ```
   numpy.array([1.4],'f',order='Fortran')
   ```

4. The 64bits precision is the default float in python. To create an *R8* array, you can use:

   ```
   numpy.array([1.4],order='Fortran')
   ```

5. The array has to be created as a char multi-dimensionnal array. An incorrect creation with a simple statement such as: `numpy.array('GoverningEquations')` produces a *wrong* zero dimension array. The correct creation for a single value could be: `numpy.array(tuple('GoverningEquations'),'|S1')` where the shape (i.e. the dimensions of the array) is `(18,)`.

CHAPTER

TWO

SPECIFIC CGNS/PYTHON TOPICS

2.1 The CGNSTree_t type

The tree structure of a CGNS data set is broken by the exception of the root node. We take the opportunity of this new CGNS/Python mapping to add a consistent root node for the CGNS tree [1].

The *CGNSTree_t* type is a node with the pattern:

```
root= [ <name:string>, None, [ <CGNSLibraryVersion:node>, <CGNSBase:node>* ], 'CGNSTree_t' ]
```

The children list is the CGNS/ADF-like root node. The *CGNSTree* node has a user-defined name, no value and a fixed *CGNSTree_t* type.

2.2 Legacy CGNS types alternative

The CGNS/SIDS defines all CGNS types and has a rule to suffix them with `_t`. There are some exceptions where some CGNS/SIDS types have been translated into strings with a special syntax.

The CGNS/Python mapping allows the use of alternate types for these, the user can either use the legacy type or the alternate CGNS/Python type. The alternate types are:

CGNS/SIDS type	CGNS/Python optional type
`"int[1+...+IndexDimension]"`	`DiffusionModel_t`
`"int[IndexDimension]"`	`Transform_t`
`"int[IndexDimension]"`	`InwardNormalIndex_t`
`"int"`	`EquationDimension_t`

Please note the `["]` character which is part of the CGNS legacy type.

Warning: This CGNS/Python feature adds *NON-SIDS* type(s) and this should be added or removed by the user application during the read and the write to the disk with a CGNS/ADF or CGNS/HDF5 compliance. The CGNS.MAP module has an option to check and remove these alternate types. As long as your application has interoperability with another CGNS/Python application there should be no problem.

2.3 Links

The **links** are used to set and get CGNS symbolic links information. This information is relevant only during read/write operations on disks. A CGNS/Python tree cannot have embedded links, as this tree is a list of lists

[1] This CGNS/Python feature adds *NON-SIDS* type(s) and this should be added or removed by the user application during the read and the write to the disk with a CGNS/ADF or CGNS/HDF5 compliance.

making a link to another list is non-sense in Python [2]. The **links** list is an extra information, not embedded into the CGNS/Python tree, and only used as disk-related operations.

> **Warning:** In the case a CGNS/Python application would not like to follow a link and then to have some *missing* data in its CGNS tree, the so-called *linked-from* node has to be removed from its parent children list.

This **links** list is an unsorted list of *link-entries* with only one entry per link. A *link-entry* is an ordered list of Python string values:

> The *target directory name* is the linked-to directory name, as it would be used to open it. It should be a valid absolute/relative file path as a plain Python string or `None`.
>
> The *target file name* is the linked-to file name, as it would be used to open it. It should be a valid absolute/relative path as a plain Python string. Its path-prefix part and its file extension part can be empty but the filename itself cannot.
>
> The *target node name* is the linked-to node name as a plain Python string. It should be the **absolute** path of the node in the linked-to file. This value cannot be empty.
>
> The *local node name* is the **absolute path** of the node in the source Python/CGNS tree. This plain Python string cannot be empty.

The links with a second level file, in other words the links in a file you are parsing after following a first link, are **always** referred as if you where in the *target filename*. Then, a list of links can be reused from one parse to another, because the `links` list is relative to the target file. The example hereafter can be an *input* as well as an *output* links list, an application would set it for a *save* or get it from a *load*:

```
[ ['/tools/CFD/ref#M6','M6_A.cgns'  ,'/Base#1/ReferenceState',
                                     '/Base/ReferenceState'],
  ['/tmp/restart'     ,'M6#001.cgns','/Base#1/Zone1/FlowSolution#EndOfRun',
                                     '/Base/Zone1/FlowSolution#Init']
]
```

The *target directory name* information is distinct to the filename, because you can have different actual target files depending on the search paths you set. This information is relevant as ouput from the read of an actual file, it should be set to *None* or ignored for a write. During a write, the only information taken into account should be the *target file name*, *target node name* and the *local node name*.

In the example above, the entries are interpreted in a different way depending if they are result of a *read* or directives for a *write*. In the case of a read, the first entry means that the file we have read has a node `/Base/ReferenceState` which is a link to the node `/Base#1/ReferenceState` in the file `M6_A.cgns`. The first directory of the file search path in which the file `M6_A.cgns` has been found is `/tools/CFD/ref#M6`. In the case of a *write*, the same entry means that the application should create a link for the node `/Base/ReferenceState` when it reaches it. This link would have `M6_A.cgns` as target file and `/Base#1/ReferenceState` as target node. The `/tools/CFD/ref#M6` value is ignored.

> **Warning:** The links list is relative to the current tree. If you want to track links of links your application has to manage this by itself, setting or getting links list during the different tree traversals.

2.4 C API

There is no requirement on the way you would create or manage a numpy array at the C API level. But you have to remember that the definition of the node contents is SIDS-to-ADF which states that data arrays and index ordering use the *Fortran* convention.

You can manage all your numpy arrays with the C order in memory, but you have to be sure that the storage on the disk, i.e. using ADF or HDF5, has the correct fortran orders. The storage also has to be contiguous in the memory.

[2] A Python list is a reference, if you put a list as a child of another list the Python interpreter actually refers to the child list. Then a child can be shared by two different lists if you do not ask for a copy. In other words, the links are the natural way of referencing to lists in Python.

When you create or obtain a copy of a numpy array you can set a flag to force a C or Fortran ordering: one of the `NPY_CCONTIGUOUS` or `NPY_FCONTIGUOUS` flag can be set. In the case of a `NPY_CCONTIGUOUS` flag set, it is up to the application to set a Fortran memory layout and a Fortran index ordering while reading/writing data to/form a CGNS/ADF or CGNS/HDF5 file [3].

The *numpy C API* allows the share of memory zone. In other words you can have a *Fortran* or *C* array you can directly set as your *numpy* array without duplication. You can reduce the memory use when your application can handle this, you can also set the `NPY_OWNDATA` flag to indicate to *numpy* that it should not release the array memory when the *numpy* array object is garbaged.

[3] For example, *CGNS.MAP* detects th `NPY_FCONTIGUOUS` and forces a data and dimensions transpose during the read/write (unless the user forces the *CGNS.MAP.S2P_NOTRANSPOSE* flag in the `load` or the `save`).

CHAPTER

THREE

EXAMPLES AND TIPS

Python comes from the C world, as well as the numpy library. This means that many behavior are assuming C-order in dimensions. The CGNS/Python mapping states that arrays should have a Fortran indexing for their actual data and that the dimension order of the data is those detailled in the [CG2] and [CG3] documents.

We give here some known issues and tips to handle this Fortran indexing in CGNS/Python. We use specific CGNS/SIDS structures to illustrate our examples.

3.1 IndexRange_t

The *IndexRange_t* is an integer array of dimensions *(IndexDimensions,2)* as detailled in [1]. The node data, in the example here, is two points with three indices. The *Python-ish* way to define them is to have a list of two lists of integers, which leads to problems if you forget your fortran order. We want to set a node with the following Python code:

```
node=['PointRange', a, [], 'IndexRange_t']
```

Now we see how to declare a correct *a* variable as a *numpy* array. If you do not specify an order to *numpy*, the default is the C-order:

```
>>> a=numpy.array([[1,2,3],[4,5,6]],dtype=numpy.int32)
>>> numpy.isfortran(a)
False
>>> a[0]
[1,2,3]
>>> a.shape
(2,3)
```

This *numpy* array is correct but you would have to transpose dimensions are memory layout before a storage on disk. Or you can enter the list itself using an explicit Fortran-order:

```
>>> a=numpy.array([[1,4],[2,5],[3,6]],dtype=numpy.int32)
>>> numpy.isfortran(a)
False
>>> a[0]
[1,4]
>>> a.shape
(3,2)
```

In that case, the *shape* is correct but the user has no mean to know wether your convention is C or Fortran. You can set the fortran flag for this. The possible creation of the array above is then:

```
>>> a=numpy.array([[1,4],[2,5],[3,6]],dtype=numpy.int32,order='Fortran')
>>> numpy.isfortran(a)
True
>>> a[0]
[1,4]
```

```
>>> a.shape
(3,2)
```

Then an application can detect your array has *Fortran* order and should be stored as found without any transpose.

3.2 IndexArray_t

There is another example switching from one order to another, this is used to add a point in a list in an easier way

```
node=['PointList', a, [], 'IndexArray_t']
```

The possible creation of the array `a` above is then:

```
>>> a=numpy.array([[1,4],[2,5],[3,6]],dtype=numpy.int32,order='Fortran')
>>> a
array([[1, 4],
       [2, 5],
       [3, 6]], dtype=int32)

>>> a=numpy.array(a.T.tolist()+[[7,8,9]],dtype=numpy.int32,order='Fortran').T
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]], dtype=int32)
```

You see that the syntax is completely unreadable, we use the *numpy* transpose attribute `T` to switch from *Fortran* to *C* order and back.. If you start with the *C* order, the Python syntax is clear:

```
>>> a=numpy.array([[1,2,3],[4,5,6]],dtype=numpy.int32)
>>> a
array([[1,2,3],
       [4,5,6],  dtype=int32)

>>> a=numpy.array(a.tolist()+[[7,8,9]],dtype=numpy.int32)
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]], dtype=int32)
```

And the application in charge of the *write* to the disk that would detect the abscence of *Fortran* flag and then transpose the array and its dimension.

3.3 DimensionalUnits_t

This node contains strings. The strings are an issue in CGNS/Python because we want to use the raw level for *numpy* (instead of *numpy* module proposed for string manipulation). We want to keep a common interface for all nodes and we do not want an exception with strings. The *DimensionalUnits_t* node can be defined as:

```
node=['DimensionalUnits', a, [], 'DimensionalUnits_t']
```

Now we see how we can defined the *numpy* array in variable *a*. The *DimensionalUnits_t* states we need a *(32,5)* array of chars. In the case of a fixed size multi-dimensionnal string array, each entry should be split as a sequence with a fixed max size (usually 32 chars):

```
a=numpy.array([
               tuple('%-32s'%'Kilogram'),
               tuple('%-32s'%'Meter'),
               tuple('%-32s'%'Second',)
               tuple('%-32s'%'Kelvin'),
```

```
    tuple('%-32s'%'Radian'),
  ],'|S32',order='Fortran').T
```

The shape of the resulting array is `(32,5)` again note the `T` at the end of the command which produces the transpose. You can use a `S32`, `|S1` or `c` type directive. An important point in this *string* as an array is the trailing *spaces* you have to fill the array cell. You have to use a `string.strip` before any string operation unless your Python application is aware of this *forced* size.

3.4 Zone_t

There we have an interesting example with the use of a data of a node. The *Zone_t* node has the dimensions of the *zone*. These dimensions are a data and theses data values should be used as *dimension* attribute of the children nodes. In other words, the user takes the *Zone_t* dimensions and creates a *numpy* array with them:

```
zonenode=['Zone001',zonedims,zonechildrenlist,'Zone_t']
```

The `zonedims` *numpy* array can b set as:

```
zonedims=numpy.array([[3,2,0],[5,4,0],[7,6,0]],dtype=numpy.int32,order='Fortran')
```

in the case of a 3D structured zone with `(ni,nj,nk)=(3,5,7)`. If you want to create a solution array with these dimensions, you can to use the following syntax:

```
zonevertexsize=zonedims[:,0]
zonecellsize=zonedims[:,1]
zonevertexboundarysize=zonedims[:,2]
```

This numpy syntax allows the user to take the whole column as a so-called *slice*.

3.5 Sub-tree imports

For example, the following snippet *imports* a truncated *ReferenceState*:

```
import numpy

refvalues=[
            ['Mach',numpy.array([0.2]),[],'DataArray_t']
            ['Reynolds',numpy.array([23300000.0]),[],'DataArray_t']
            ['LengthReference',numpy.array([0.5]),[],'DataArray_t']
            ['Density',numpy.array([1.22524863848]),[],'DataArray_t']
]

data=['ReferenceState',None,refvalues,'ReferenceState_t']
```

Once *import*-ed, your Python code can insert this node in its structure (here our previous code snippet is in the file `refstate.py`:

```
import numpy
import restate

tree=['CGNSTree',None,[],'CGNSTree_t']
base=['Fuselage',numpy.array([3,3],dtype=numpy.int32),[],'CGNSBase_t']

tree[2].append(base)
base[2].append(restate.data)
```

3.6 Sub-tree share

A list is a reference. If you put a list into another one, you do not perform a copy, you use a reference. Then the modification of the first list is in the second:

```
>>> a=[1,2,3]
>>> b=[a,[4,5,6]]
>>> b
[[1, 2, 3], [4, 5, 6]]

>>> a[1]=9
>>> b
[[1, 9, 3], [4, 5, 6]]
```

You always have to take care of the lists, in particular if you use large CGNS/Python trees you want to share to optimize memory. Another point to keep in mind is that *numpy* copies do *NOT* propagate *Fortran* flag.

3.6.1 Glossary

cgns.org In this document, *cgns.org* refers to the official CGNS web site and by extension to its contents. For example, the *cgns.org* documentation is the official documentation found on this web site. *CGNS* stands for *CFD General Notation System*.

CGNS/SIDS The specification of the CGNS data model. This *cgns.org* document is the reference for the specification of a *CGNS* compliant tree at the *conceptual* level. The implementation is achieved once a *mapping* has been selected (e.g. CGNS/ADF, CGNS/HDF or CGNS/Python). *SIDS* stands for *Standard Interface Data Structures*.

CGNS/MLL The implementation of the CGNS/ADF and CGNS/HDF specifications. This librarie and its C and Fortran APIs are available on *cgns.org* web site. *MLL* stands for *mid-level library' (ADF and HDF5 are the 'low-level libraries*).

CGNS/ADF The *cgns.org* mapping document describing the implementation of SIDS on the ADF storage layer. This doesn't include the C or fortran actual implementation which is available only in the CGNS/MLL librarie. *ADF* stands for *Advanced Data Format* developed by *Boeing* and *NASA*.

CGNS/HDF The *cgns.org* mapping document describing the implementation of SIDS on the HDF5 storage layer. This doesn't include the C or fortran actual implementation which is available only in the CGNS/MLL librarie. *HDF* stands for *Hierarchical Data Format* developed by the *hdf group*.

Python Python is a programming langage. Its success comes from its easy and powerful syntax and its capabilities in being extended.

numpy The most used numerical library for the Python programming langage.

BIBLIOGRAPHY

[CG1] CGNS SIDS - Standard Interface Data Structure http://www.grc.nasa.gov/WWW/cgns/sids

[CG2] SIDS-to-ADF Mapping Reference Manual http://www.grc.nasa.gov/WWW/cgns/filemap

[CG3] SIDS-to-HDF Mapping Reference Manual http://www.grc.nasa.gov/WWW/cgns/filemap_hdf

[PY1] Python Programming langage http://www.python.org

[PY2] Numpy http://numpy.scipy.org