AIAA-2012-1264

50th AIAA Aerospace Sciences Meeting, January 9 – 12, 2012, Nashville, TN

Recent Updates to the CFD General Notation System (CGNS)

Christopher L. Rumsey*

NASA Langley Research Center, Hampton, VA 23681

Bruce Wedan†

Computational Engineering Solutions, San Ramon, CA 94582

Thomas Hauser‡

University of Colorado Boulder, Boulder, CO 80309

Marc Poinot§

ONERA - The French Aerospace Lab, F-92322 Chatillon, FRANCE

The CFD General Notation System (CGNS)—a general, portable, and extensible standard for the storage and retrieval of computational fluid dynamics (CFD) analysis data—has been in existence for more than a decade (Version 1.0 was released in May 1998). Both structured and unstructured CFD data are covered by the standard, and CGNS can be easily extended to cover any sort of data imaginable, while retaining backward compatibility with existing CGNS data files and software. Although originally designed for CFD, it is readily extendable to any field of computational analysis. In early 2011, CGNS Version 3.1 was released, which added significant capabilities. This paper describes these recent enhancements and highlights the continued usefulness of the CGNS methodology.

Glossary of Terms

ADF	CGNS Advanced Data Format, www.grc.nasa.gov/www/cgns/CGNS_docs_current/adf
API	Application Programming Interface
CGIO	CGNS low-level library, www.grc.nasa.gov/www/cgns/CGNS_docs_current/cgio
CHLone	CGNS special purpose C library (for HDF5 files only), chlone.sourceforge.net
CGNS	CFD General Notation System, cgns.org
CGNSTalk	CGNS user forum, lists.nasa.gov/mailman/listinfo/cgnstalk
CMake	Cross-platform build system, cmake.org
CPEX	CGNS Proposals for Extension, cgns.sourceforge.net/Proposals.html
GNU	UNIX-style operating system, gnu.org
HDF5	Hierarchical Data Format, hdfgroup.org/HDF5
I/O	Input/Output
MAP	Special purpose module in pyCGNS, pycgns.sourceforge.net/MAP/readme.html
MLL	CGNS Mid-Level Library, www.grc.nasa.gov/www/cgns/CGNS_docs_current/midlevel
MPI	Message Passing Interface, mcs.anl.gov/mpi
MPI4py	MPI library for Python, code.google.com/p/mpi4py
NumPy	Scientific computing package for Python, numpy.scipy.org
pyCGNS	CGNS Python package, pycgns.sourceforge.net
Python	Programming language, python.org
SIDS	CGNS Standard Interface Data Structures, www.grc.nasa.gov/www/cgns/CGNS_docs_current/sids
Tcl/Tk	Tool Command Language and Toolkit, tcl.sourceforge.net
UNIX®	Computer operating system, unix.org

*Senior Research Scientist, Computational AeroSciences Branch, Mail Stop 128, Fellow AIAA.

†Consultant.

‡Director of Research Computing, Associate Fellow AIAA.

§Software Engineer, Computational Fluid Dynamics and Aeroacoustics Department, Member AIAA.

I. Overview of CGNS

The CFD General Notation System (CGNS) originated in 1994 as a joint effort between Boeing and NASA, and has since grown to include many other contributing organizations worldwide. It is an effort to standardize CFD input and output, including grid (both structured and unstructured), flow solution, connectivity, boundary conditions, and auxiliary information. CGNS is an AIAA Recommended Practice.[1] In 1999, control of CGNS was transferred to a public forum known as the CGNS Steering Committee. This Steering Committee is made up of international representatives from government and private industry. All CGNS software is free and open to anyone (open source, under the zlib/libpng license[a]).

All CGNS documentation is on-line. This is because each is considered as a "living document" and is kept up-to-date as the software and methodology expands. Every so often, referenceable documents concerning CGNS are published as well.[1–9] However, the up-to-date on-line documentation is the best way to learn the system and remain current. The main website address for CGNS is cgns.org, and all CGNS documentation is accessible via that site. The reader is also referred to the Glossary of Terms (above) for website addresses of particular documents and applications, current as of the writing of this paper.

CGNS is easily extensible, and allows for user-inserted comments. It creates binary files that are portable across computer platforms, employing either the Advanced Data Format (ADF, part of the standard CGNS software release) or the Hierarchical Data Format (HDF5, available separately) as a data layer. CGNS also provides a layer of software known as the CGIO Interface, which allows access to these files at a low-level, and a second layer of software known as the Mid-Level Library (MLL), which standardizes and simplifies the implementation of CGNS into existing CFD codes. A Python mid-level access, pyCGNS, is also available separately.

The CGNS "standard" is described by the Standard Interface Data Structured (SIDS). The SIDS is the heart of CGNS; it defines the standards or conventions for what is meant by a grid point, flow solution, etc., and also describes how to organize the information in the file. The SIDS "file mappings" in turn map the SIDS conventions onto either an ADF or HDF5 file. Fig. 1 outlines the various ways that a user code or application can currently access a CGNS ADF or HDF5 file. Any application can read or write an ADF or HDF5 file directly via ADF or HDF5 libraries, as indicated by the dashed arrows in the figure. Alternatively, the new CGIO interface improves upon this low-level access, because it has the same functionality but is independent of the file type (ADF or HDF5). When using ADF, HDF5, or CGIO libraries, it is up to the individual application to adhere to the SIDS standard. On the other hand, the mid-level access software "understands" the SIDS file mapping, so use of MLL or pyCGNS[b] insures full compliance with the SIDS. The fact that multiple implementations are possible is evidence of the flexibility of the CGNS standard and its mappings.

A CGNS file (sometimes referred to as a database) is an entity that is organized into a set of "nodes" in a tree-like structure, in much the same way as directories are organized in the UNIX®environment. An example of a CGNS tree-like structure is shown in Fig. 2. The top-most node is referred to as the "root node." Each node below the root node is defined by both a name and a label, and may or may not contain information and/or data. Each node can also be a "parent" to one or more "child" nodes. A node can also have as a child node a link to a node elsewhere in the file or to a node in another CGNS file. Links are transparent to the user in that the data and any child nodes are accessed in the same way as standard nodes.

Two sample layouts for a structured and unstructured version of the same grid are given in Figs. 3(a) and (b). (In Fig. 3(b) the `CoordinateY` and `CoordinateZ` nodes have been omitted for clarity.) Note that CGNS allows a grid to be made up of *both* structured and unstructured zones in the same file (i.e., a mixed type), if desired. Additional details about the fundamental aspects of CGNS, including usage of many of the basic MLL routines, can be found in the User's Guide to CGNS.[2]

CGNS uses an extension process called CPEX, to allow any user to propose extensions to the standard. The CPEX process includes review by the CGNS Steering Committee. If accepted, the extension is subsequently integrated into the SIDS and the MLL.

II. Utilization of CGNS

Since its inception, CGNS has been actively adopted by many groups world-wide. Although it has typically operated unfunded over the last 10 years, the CGNS Steering Committee has remained an active organization, overseeing

[a] opensource.org/licenses/zlib-license.php, cited: 11/3/2011.

[b] As will be mentioned later in the Python Mapping section, pyCGNS can also make use of CHLone (not shown in the figure). CHLone is a recently-developed special-purpose CGNS library, available separately for use with HDF5.

the direction and activities of the CGNS definitions, software development, and maintenance. CGNS tutorial sessions were held in conjunction with AIAA meetings in San Francisco in 2006 and in Orlando in 2010, and also at a European Conference for Aero-Space Sciences (EUCASS) meeting in Paris in 2009. Tutorials are also available on the CGNS website. The current membership of the CGNS Steering Committee is shown in Table 1.

Table 1. CGNS Steering Committee members as of November 2011.

Organization	Representative
ADAPCO	Steve Feldman
Airbus	Renaud Sauvage
ANSYS-CFX	Richard Hann
ANSYS-ICEM CFD	Simon Pereira
Boeing Commercial	Will Stoffers
Computational Engineering Solutions	Bruce Wedan
Concepts NREC	Mark Anderson
Intelligent Light	Earl Duque
NASA Glenn	Tony Iannetti
NASA Langley	Chris Rumsey
ONERA	Marc Poinot
Pointwise, Inc.	John Chawner
Pratt & Whitney	Bob Bush
Rolls-Royce Allison	Leigh Lapworth
Stanford University	Juan Alonso
Stony Brook University	Xiangmin Jiao
Tecplot, Inc.	Scott Imlay
TTC Technologies	Ken Alabi
University of Colorado	Thomas Hauser (current Chair)
US Air Force / AEDC	Greg Power

CGNS is supported primarily by its users, through a self-help e-mail exchange of information called CGNSTalk. As of November 2011, the cite had nearly 350 registered users from 22 different countries. CGNS has been implemented in many different software packages; examples are listed on the CGNS website.

CGNS has also proved to be very useful for the productive and rapid exchange of CFD grids at the AIAA Drag Prediction series of workshops[10] and at the AIAA High Lift Prediction Workshop.[11] ONERA (the French Aerospace Lab) has adopted CGNS as a key data model for its CFD-based workflow process.[12] In fact, CGNS is now the main file format used for large simulation data exchange of most of ONERA's CFD projects. As described in the reference listed, the five reasons for ONERA moving toward the CGNS hierarchical data format were its: (1) tree structure, (2) numerical-simulation-aware middleware, (3) portability, maintainability, compatibility, (4) availability (open system), and (5) trustworthiness and support.

III. Recent Enhancements to CGNS

Recent enhancements to CGNS are described below. Referring to Fig. 1, the following sections are organized from the bottom up. First, changes at the file level are described, followed by changes in low-level access, SIDS, and mid-level access. Finally, new applications are described.

A. File-Level Changes

1. Official Adoption of HDF5

CGNS was originally built using the ADF file format. This format was based on a common file format system previously in use at McDonnell Douglas. The ADF has worked extremely well and has required little repair, upgrade,

or maintenance over the last decade.

However, ADF does not have parallel I/O or data compression capabilities, and lacks the support and tools that the HDF5 storage format offers. HDF5 has rapidly grown to become a world-wide standard for storing scientific data. HDF5 has parallel I/O capability as well as a broader support base than ADF.

Therefore, the CGNS Steering Committee made the decision to adopt HDF5 as the default (official) data storage mechanism. However, because it is possible to easily support both ADF and HDF5 formats simultaneously (ensuring backward compatibility[c] as well as giving the user a choice), both will continue to be supported indefinitely. CGNS now handles HDF5 and ADF transparently through the new CGIO interface. See the CGIO Interface section below.

If the user links to HDF5[d] when compiling CGNS Version 3.1 or later, then by default the software will write HDF5 files, although the user can force the writing of ADF files instead if desired. The software will automatically read either format without the user having to do anything.

To summarize: the CGNS Steering Committee considers HDF5 to be its official, recommended storage format. However, ADF will continue to be available for use, with the CGNS mid-level library capable of (1) using either format and (2) translating back and forth between the two.

2. 64-Bit Integer Capability

A major change in CGNS Version 3.1 is the capability to use 64-bit integers. This is necessary because many indexing arrays associated with CFD grids approaching hundreds of millions or more points or elements exceed the 32-bit integer limit, and cannot be written with the earlier software. The 64-bit integer capability is handled through the introduction of data type `cgsize_t`, which is a 64-bit integer when building 64-bit code and a 32-bit integer otherwise.

The configuration options and building of CGNS are now supported by both CMake and GNU configure scripts. Some of the new options with both methods allow building 64-bit code (where applicable) and support for legacy code. Legacy refers to 32-bit CGNS versions prior to version 3.1.

A CGNS file written in 64-bit mode will only be readable by the version 3.1 software or later. A 32-bit compilation of version 3.1 will be able to read a 64-bit CGNS file, but will fail gracefully if the dimensions exceed those that can be handled by a 32-bit integer. With a 32-bit mode legacy compilation of version 3.1, the ADF and HDF5 interfaces are unchanged; and thus the CGNS files will be readable by earlier versions. Regardless of how the version 3.1 software is compiled, it is always backward compatible with earlier version CGNS files.

3. 64-Bit Portability

For C-portability between 32 and 64-bit compilation modes, new code should use the `cgsize_t` data type. If support for CGNS versions prior to 3.1 (where `cgsize_t` is not defined) is desired, then the following code (or something similar) should also be used:

```
#if CGNS_VERSION < 3100
#define cgsize_t int
#endif
```

Existing C-code that uses `int` will not work with a CGNS 3.1 library compiled in 64-bit mode. In this case, if the code is not modified to use `cgsize_t`, the following should be added to the code:

```
#if CGNS_VERSION >= 3100 && CG_BUILD_64BIT
#error does not work in 64 bit mode
#endif
```

In Fortran, all integer arguments in the interface are taken to be `integer*4` in 32-bit mode and `integer*8` in 64-bit mode. If one uses default or implicit integers, a Fortran code should port to 64-bit mode by simply turning on the compiler option that promotes implicit integers to `integer*8`. However, if the integers are explicitly defined as `integer*4`, then a Fortran code will not work in 64-bit mode.

The `cgnslib_f.h` header file has been modified to explicitly define all integer values based on the CGNS library compilation mode: `integer*8` when compiled in 64-bit mode and `integer*4` otherwise. A new integer parameter has also been added to the header, `CG_BUILD_64BIT`, which will be set to 1 in 64-bit mode and 0 otherwise. A user may use this parameter to check if the CGNS library has been compiled in 64-bit mode or not, as in:

[c] Backward compatibility—meaning that CGNS software can always read old files from earlier versions—is one of the core policies adopted by the CGNS Steering Committee. However, files that are written with newer versions are not necessarily readable by earlier versions of the software; i.e., forward, or upward, compatibility is not guaranteed.

[d] HDF5 version 1.8 or later is required for support of links.

```
if (CG_BUILD_64BIT .ne.  0) then
   stop 'CGNS will not work in 64-bit mode'
endif
```
If using a CGNS library prior to version 3.1, this parameter will not be defined.

B. Low-Level Access Change: CGIO Interface

The new CGIO routines provide low-level access to the underlying file manager (ADF or HDF5) and are used to store and retrieve the data. These routines are patterned along the lines of the original ADF core routines but work transparently to the user for both ADF and HDF5.

Like the MLL calls, the CGIO calls are available in both C and Fortran. They can be divided into several categories:

- File-Level Routines
- Data Structure Management Routines
- Link Management Routines
- Node Management Routines
- Data I/O Routines
- Error Handling and Messages
- Miscellaneous Routines

CGIO calls are very basic (at a lower level than the MLL calls), and have no inherent knowledge of the SIDS. For example, `cgio_create_node` creates an empty node, `cgio_set_label` sets its label, `cgio_set_dimensions` sets its data type and dimensions, and `cgio_write_all_data` writes its data. Details are not given here; they can be found in the on-line documentation for CGIO.

C. Standard Interface Data Structures Changes

1. Unstructured Polyhedral Elements Capability

Examples were given earlier for writing both structured and unstructured grids in CGNS. Typically for unstructured grids, the elements are explicitly defined (although possibly mixed) shapes such as hexahedra, tetrahedra, pyramids, pentahedra, etc. However, there are applications that require the use of general polyhedra of completely arbitrary shape. These arbitrary polyhedral elements can now be written in CGNS. When describing general polyhedra, two `Elements_t` nodes are required: one (with `ElementType` = `NGON_n`) to define the connectivity of the individual faces of all the polyhedral elements, and another (with `ElementType` = `NFACE_n`) to define how the faces connect to form the polyhedra.

As an example, consider Fig. 4, which is a very simple grid made up of 6 nodes and 3 tetrahedral elements. Naturally, this grid can be described most easily with one `Elements_t` node in which `ElementType` = `TETRA_4`, `ElementRange` = [1,3], `ElementConnectivity` = (1,2,3,4), (2,5,3,6), (2,6,3,4). However, it can also be described in terms of 3 general polyhedra, as follows. There are 10 unique faces, each defined in this case by 3 nodes: the connectivity for each of these would be described in an `Elements_t` node of type `NGON_n`. Then, the connectivity of the faces making up the volume elements (in this case 4 faces for each of the 3 volume elements) would be described in an `Elements_t` node of type `NFACE_n`. In this particular case, faces (1,2,3,4) make up element 1, faces (5,6,7,8) make up element 2, and faces (8,9,10,3) make up element 3. Note that by convention, face normals point outward from the element; for faces with inward-pointing normals the face numbers must be given a negative number. This example is illustrated in the tree below.

```
CGNSBase_t
   Zone_t Zone1
         GridCoordinates_t GridCoordinates
         Elements_t Elementfaces
               ElementType = NGON_n
               ElementRange = [1,10]
               ElementConnectivity =
                  (3,1,3,2), (3,1,2,4), (3,2,3,4), (3,3,1,4),
                  (3,2,3,5), (3,2,5,6), (3,5,3,6), (3,3,2,6),
                  (3,2,6,4), (3,6,3,4)
```

```
      Elements_t Elementvolumes
            ElementType = NFACE_n
            ElementRange = [11,13]
            ElementConnectivity =
               (4,1,2,3,4), (4,5,6,7,8), (4,-8,9,10,-3)
```

2. Time-Dependent Connectivities

Time-dependent connectivities, i.e., how different grid zones connect, interact, or overlap with each other, has undergone a minor enhancement as of CGNS Version 3.1. Previously, only one `ZoneGridConnectivity_t` node was allowed per zone. So, for time-dependent solutions, connectivity had to remain fixed.

In the new implementation (defined on-line as CPEX 0027) multiple instances of `ZoneGridConnectivity_t` are now allowed. Furthermore, under `ZoneIterativeData_t`, an optional DataArray `ZoneGridConnectivityPointers` was added for associating the various connectivities with particular times. This fixes the earlier limitation, and now allows for time-dependent connectivity information.

An example tree is given below for a situation with one stationary zone and one zone with rigid grid motion. The connection information between the two zones varies with time, so multiple `ZoneGridConnectivity_t` nodes are necessary in each zone.

```
CGNSBase_t
   BaseIterativeData_t
         NumberOfSteps = N
         TimeValues = time1, time2, ...  , timeN
   Zone_t StationaryZone
         GridCoordinates_t GridCoordinates
         ZoneBC_t ZoneBC
         ZoneGridConnectivity_t Con1
         ZoneGridConnectivity_t Con2
         ...
         ZoneGridConnectivity_t ConN
         FlowSolution_t Soln1
         FlowSolution_t Soln2
         ...
         FlowSolution_t SolnN
         ZoneIterativeData_t
               ZoneGridConnectivityPointers = Con1, Con2, ...  , ConN
               FlowSolutionPointers = Soln1, Soln2, ...  , SolnN
   Zone_t MovingZone
         GridCoordinates_t GridCoordinates
         ZoneBC_t ZoneBC
         RigidGridMotion_t Rig1
         RigidGridMotion_t Rig2
         ...
         RigidGridMotion_t RigN
         ZoneGridConnectivity_t Con1
         ZoneGridConnectivity_t Con2
         ...
         ZoneGridConnectivity_t ConN
         FlowSolution_t Soln1
         FlowSolution_t Soln2
         ...
         FlowSolution_t SolnN
         ZoneIterativeData_t
               RigidGridMotionPointers = Rig1, Rig2, ...  , RigN
               ZoneGridConnectivityPointers = Con1, Con2, ...  , ConN
               FlowSolutionPointers = Soln1, Soln2, ...  , SolnN
```

3. General SIDS Improvements

General SIDS improvement involved several different changes, described on-line in CPEX 0031. Here, two of the most significant changes are highlighted:

- Use of new `FamilyBCDataSet_t` under `FamilyBC_t`. Previous use of `BCDataSet_t` was inconsistent; the new `FamilyBCDataSet_t` structure is more appropriate. Its intended use is for simple boundary-condition types (Dirichlet and Neumann data), where the equations imposed do not depend on local flow conditions.
- Exclusive use of `PointRange` or `PointList`. These indexing names are now used to define the range or list of vertices, edges, faces, or cells (elements) within `FlowSolution_t`, `DiscreteData_t`, and `BC_t`. They are also used in the new `ZoneSubRegion_t` node, as described in the next section below. The indexing names `ElementRange` and `ElementList` are no longer used, but can still be read by the MLL software (the software automatically switches to the new usage when the old usage is detected).

For the latter change in `FlowSolution_t` and `DiscreteData_t`, `PointRange` and `PointList` refer to vertices, edges, faces, or cells (elements), depending on the value assigned to `GridLocation`, as shown in Table 2. For `BC_t`, `PointRange` and `PointList` list the vertices, edges, or faces according to Table 3. In these tables, `*FaceCenter` stands for the possible types: `IFaceCenter`, `JFaceCenter`, `KFaceCenter`, or `FaceCenter`.

Table 2. `GridLocation` meanings for `FlowSolution_t` or `DiscreteData_t` data.

CellDimension	GridLocation			
	Vertex	EdgeCenter	*FaceCenter	CellCenter
1	vertices	-	-	cells (line elements)
2	vertices	edges	-	cells (area elements)
3	vertices	edges	faces	cells (volume elements)

Table 3. `GridLocation` meanings for `BC_t` data.

CellDimension	GridLocation		
	Vertex	EdgeCenter	*FaceCenter
1	vertices	-	-
2	vertices	edges	-
3	vertices	edges	faces

4. Regions

The new capability of Regions (described on-line as CPEX 0030) had been identified as a need in CGNS for some time. Previously, CGNS only allowed flowfield or other information associated with the grid to be given over the entire grid. Regions now adds the ability to give flowfield or other information over a *subset* of the entire zone in a CGNS file. This subset may be over a boundary, a portion of a boundary, or a portion of the volume. The region is defined via the new node specification `ZoneSubRegion_t`, which is a child node of `Zone_t`.

An example of a typical use of the new Regions capability is shown in Fig. 5. The additional `ZoneSubRegion_t` and its children are located under `Zone_t`. In this particular example, it is desired to store a subregion of boundary temperatures at specific boundary elements only. Because the subregion is only on the boundary of a 3-D dataset, it is a topologically 2-D surface region with `RegionCellDimension` = 2. The GridLocation specification of FaceCenter indicates that the `PointRange` is referring to the face centers of the 2-D boundary elements (i.e., the temperature is stored at the face centers of elements numbered 2561–2688, inclusive). `GridLocation` meanings for Regions are

summarized in Table 4. The option to inherit the boundary information (list length and element list) from an existing BC_t node is also allowed, by specifying the name of the node under RegionName (Descriptor_t).

Regions can be used to store any type of 1-D, 2-D, or 3-D data that users may deem necessary or useful for their application. For instance, users at ONERA make use of ZoneSubRegion_t for data storage at the connectivity level; in this case the use of Regions allows a more formal mechanism for storing additional non-abutting interpolation information, instead of requiring the use of generic UserDefined data in the connectivity node.

Table 4. GridLocation meanings for ZoneSubRegion_t data.

Cell-Dimension	Region-CellDimension	GridLocation Vertex	EdgeCenter	*FaceCenter	CellCenter
1	1	vertices	-	-	cells (line elements)
2	1	vertices	edges	-	-
2	2	vertices	edges	-	cells (area elements)
3	1	vertices	edges	-	-
3	2	vertices	edges	faces	-
3	3	vertices	edges	faces	cells (volume elements)

D. Mid-Level Access Changes

1. *New MLL Functions*

Many new MLL functions have been added to the CGNS library recently. These are summarized in Table 5.

2. *Parallel CGNS MLL*

Also new with CGNS Version 3.1 is the alpha version of the parallel CGNS MLL library. The "alpha" indicates availability for testing and preliminary use, but it is not fully integrated or supported. Although not currently integrated within the CGNS library, it provides the initial framework for testing and further developing these capabilities. Previous work in this area has been done by Hauser and Pakalapati.[6,13] In an effort to improve performance and better integrate the parallel extension into the CGNS MLL, new routines were written that access HDF5 directly and take full advantage of the collective I/O support in HDF5 version 1.8—see Hauser and Horne.[9] The parallel extension is meant to be a supplement of the CGNS MLL. A single process should create the layout and structure of the CGNS file. Then the parallel library can be used to write the three-dimensional grids, solution, and other data arrays in parallel. An overview of the API interface for the parallel extension is given in Table 6. To improve performance and address shortcomings in the HDF5 API, an I/O queuing approach has also been implemented, which queues the I/O until a flush function is called, at which point the write commands are analyzed and executed with HDF5 calls. This provides MPI-IO sufficient data to effectively recognize the collective and continuous nature of the data being written.

3. *Python Mapping*

Python is now widely used for high performance computing numerical simulations as a user interface and steering language for multi-physics simulations.[14] The Python programming language has recently been implemented as a SIDS physical representation via CGNS mapping concepts. The use of Python facilitates the modification of existing programs during a complex simulation setup. The language is easy to extend. The CGNS/Python mapping adds the required common logical representation of data as well as a common way to exchange complex data structures at runtime. As a result, creating interoperability between simulation codes is straightforward.

Table 5. Recent MLL additions.

cg_elements_partial_write	write a subpart of an elements array
cg_save_as	save the open CGNS file
cg_set_file_type	set default file type
cg_get_file_type	get file type for open CGNS file
cg_error_handler	set CGNS error handler
cg_set_compress	set CGNS compression mode
cg_get_compress	get CGNS compression mode
cg_set_path	set the CGNS link search path
cg_add_path	add to the CGNS link search path
cg_boco_gridlocation_read	read boundary condition location
cg_boco_gridlocation_write	write boundary condition location
cg_sol_ptset_read	read a point set FlowSolution_t node
cg_sol_ptset_write	create a point set FlowSolution_t node
cg_sol_ptset_info	get info about a point set FlowSolution_t node
cg_sol_size	get the dimensions of a FlowSolution_t node
cg_discrete_ptset_read	read a point set DiscreteData_t node
cg_discrete_ptset_write	create a point set DiscreteData_t node
cg_discrete_ptset_info	get info about a point set DiscreteData_t node
cg_discrete_size	get the dimensions of a DiscreteData_t node
cg_nsubregs	get number of ZoneSubRegion_t nodes
cg_subreg_info	get info about a ZoneSubRegion_t node
cg_subreg_ptset_read	read point set data for a ZoneSubRegion_t node
cg_subreg_ptset_write	create a point set ZoneSubRegion_t node
cg_subreg_bcname_read	read the BC_t node name for a ZoneSubRegion_t node
cg_subreg_bcname_write	create a ZoneSubRegion_t node that references a BC_t node
cg_subreg_gcname_read	read GridConnectivity_t node name for a ZoneSubRegion_t node
cg_subreg_gcname_write	create ZoneSubRegion_t node that references GridConnectivity_t
cg_cell_dim	get the cell dimension for the CGNS base
cg_index_dim	get the index dimension for the CGNS zone
cg_nzconns	get number of ZoneGridConnectivity_t nodes
cg_zconn_read	read ZoneGridConnectivity_t node
cg_zconn_write	create ZoneGridConnectivity_t node
cg_zconn_set	set the current ZoneGridConnectivity_t node
cg_zconn_get	get the current ZoneGridConnectivity_t node

Table 6. API of the Parallel CGNS MLL as implemented in `pcgnslib.c` and declared in `pcgnslib.h`. Software using the library to access CGNS files in parallel must use the routines listed here.

General File Operations	
`cgp_open`	Open a new file in parallel
`cgp_base_read`	Read the details of a base in the file
`cgp_base_write`	Write a new base to the file
`cgp_nbases`	Return the number of bases in the file
`cgp_zone_read`	Read the details of a zone in the base
`cgp_zone_type`	Read the type of a zone in the base
`cgp_zone_write`	Write a zone to the base
`cgp_nzones`	Return the number of zones in the base
`cgp_nsols`	Read the number of solutions in a zone
`cgp_close`	Close a file
Coordinate Data Operations	
`cgp_coord_write`	Create the node and empty array to store coordinate data
`cgp_coord_write_data`	Write coordinate data to the zone in parallel
`cgp_coord_read_data`	Read coordinate data in parallel
Unstructured Grid Connectivity Operations	
`cgp_section_write`	Create the nodes and empty array to store grid connectivity for an unstructured mesh
`cgp_section_write_data`	Write the grid connectivity to the zone in parallel for an unstructured mesh
`cgp_section_read_data`	Read grid connectivity data in parallel
Solution Data Operations	
`cgp_sol_write`	Create the node and empty array to store solution data
`cgp_sol_write_data`	Write solution data to the zone in parallel
`cgp_sol_read_data`	Read solution data in parallel
General Array Operations	
`cgp_array_write`	Create the node and empty array to store general array data
`cgp_array_write_data`	Write general array data to the zone in parallel
`cgp_array_read_data`	Read general array data in parallel
Queued I/O Operations	
`queue_slice_write`	Queue a write operation to be executed later
`queue_flush`	Execute queued write operations

A CGNS/Python mapping sets a CGNS node as a Python list, with a node name, a node value (a NumPy array), a list of children, and a CGNS type:

```
<CGNS/Python-node> = [  <node-name:string>,
                        <node-value:NumPy array>,
                        [ <children:  ListOf<CGNS/Python-node> ],
                        <node-sidstype:string>
                        ]
```

For example, below is an example of a boundary condition definition with a CGNS/Python mapping:

```
BC1=  ['Wall', array(['B', 'C', 'W', 'a', 'l', 'l'], dtype='|S1'),
      [ ['PointRange', array([[ 1, 37],[ 1, 1],[ 1, 2]], dtype=int32),
      [], 'IndexRange_t']],
      'BC_t']
```

A complete CGNS tree can be defined as a Python list of lists with only raw Python types and NumPy arrays. A CGNS user can define the CGNS data tree with open source libraries (Python and NumPy) and can define the interface to a Python module using this mapping. The use of this Python mapping can be illustrated with three examples:

1. Partial CGNS tree manipulation: Parallel multi-physics simulations often handle very large data—most of the time within a specific tool in the workflow—on a process or in a thread that needs access to a part of a complete CGNS tree. The CGNS/Python mapping allows loading or saving a part of the tree while still having a compliant structure. For this purpose we use the pyCGNS Python package. It has a MAP module that has only two functions: the load and the save. The first reads a part of a CGNS HDF5 file and creates the corresponding CGNS/Python mapping while the second performs a save of a part of a CGNS/Python tree in a CGNS HDF5 file. The pyCGNS.MAP module uses CHLone, which accesses a node only when the application requires it. For example, in a rigid motion of a block, the connectivity may change. Then the application adds the new "connected-to" zone to its CGNS/Python tree without also loading the coordinates of the zone points. This dynamic approach leads to manipulation of partial trees with the minimum possible memory footprint.

2. Interoperability and code-coupling: There are three main strategies for data interoperability at runtime: pass information between codes with (1) a file, (2) a network connection, or (3) a shared memory. File interoperability is insured by the CGNS HDF5 mapping. Both network and memory interoperability of data can be insured by using CGNS/Python. Two codes can inter-operate in the same Python script by passing a complete CGNS/Python tree in memory. The codes can be either C or Fortran software, and a memory chunk representing an actual array in memory is shared (as far as the two codes have a shared memory service available). The NumPy API allows the passing of direct pointers to memory instead of copying the data. The array even has a so-called "fortran" flag that gets triggered if the memory chunk is coming from a Fortran column-major software as opposed to a C row-major software. This makes it possible to create an array in Python, pass it to a Fortran solver, retrieve it again in Python, and finally pass it to a C code for post-processing. The Python objects, including NumPy arrays, have serialization/de-serialization functions in order to pass information to the network (or MPI for example). A whole CGNS/Python tree can be exchanged from one code to another with a simple call to an MPI Python library such as MPI4py. This can be used to provide data interoperability using the network.

3. Easy pre/post-processing: The ease of use of Python and the powerful set of NumPy functions allow for fast and concise pre- and post-processing of numerical simulation data. The NumPy has vectorized functions working the actual array memory. This, together with recursive parsing of the tree structured CGNS/Python mapping, leads to short, reusable, and maintainable sets of CGNS pre- or post-processing utilities (such as block splitting and scatter/gather, data on-the-fly extraction, use of network communication script between codes, etc.). As the main data used in the simulation workflow is a CGNS/Python tree, any application uses SIDS to get or set the data in the tree that goes from one code to another with the same data model and a common physical representation.

Although CGNS/Python is not well-fitted for time-consuming computations, it is extremely useful for simulation software assembly in multi-physics workflow processes. Currently, it is possible to conduct large high-performance multi-physics simulations with a Python management of workflow, data, and combinations of C/C++/Fortran codes.[15]

E. New Applications

CGNSview is a relatively new tool in the CGNS software package. It is a CGNS file viewer and editor based on Tcl/Tk. CGNSview replaces a previous tool known as ADFviewer, and is now based on the CGIO interface which reads both ADF and HDF5 files. The graphical user interface allows access to any node in the file using a collapsible node tree. Nodes and data may be added, deleted, and modified. An example screen shot of the CGNSview interface is shown in Fig. 6. Several other utilities may be accessed from CGNSview, including: (1) CGNSplot - for displaying the mesh, etc., (2) CGNScalc - a calculator using data in the CGNS file, (3) CGNScheck - a CGNS file validator, (4) CGNSversion - changes the version number for any CGNS file, and (5) import, export, data conversion, and subset extraction and interpolation utilities.

IV. Concluding Remarks

CGNS is a standard for storing CFD grids and simulation data. This paper has described recent enhancements to this standard and the supporting software ecosystem. Over the years, CGNS has proved to be long-lasting and stable, yet readily extensible to handle new types of data. In particular, with its policy of always maintaining backward-compatibility, CGNS has established a trustworthiness in its longevity as a storage medium.

The recent upgrade to allow the use of 64-bit integers has extended CGNS's capability to handle grids of any practical size in today's working environment. With 32-bit integers, grid sizes were previously limited to the 10s or 100s of millions of grid elements, depending on the type of elements. In addition to the usual structured and unstructured elements, the CGNS methodology now allows for the storage of arbitrary unstructured polyhedral elements.

CGNS capabilities continue to grow. As more people adopt it and its support base is enhanced, additional functionality required for large multi-physics simulations can be incorporated. Already, CGNS has been useful not only for individuals (providing a stable and descriptive CFD storage method), but also for groups and workflows where collaboration and rapid exchange of CFD grids and solutions helps to foster a competitive advantage.

Acknowledgments

The Subsonic Fixed Wing Project of NASA's Fundamental Aerodynamics Program supported some of this work.

References

[1]CGNS Steering Committee, "The CFD General Notation System Standard Interface Data Structures," AIAA R-101A-2005, American Institute of Aeronautics and Astronautics, Reston, VA, 2007.

[2]Rumsey, C. L., Poirier, D. M. A., Bush, R. H., and Towne, C. E., "A User's Guide to CGNS," NASA/TM-2001-211236, October 2001.

[3]Poirier, D. M. A., Allmaras, S. R., McCarthy, D. R., Smith, M. F., and Enomoto, F. Y., "The CGNS System," AIAA Paper 98-3007, 1998.

[4]Poirier, D. M. A., Bush, R. H., Cosner, R. R., Rumsey, C. L., and McCarthy, D. R., "Advances in the CGNS Database Standard for Aerodynamics and CFD," AIAA Paper 2000-0681, 2000.

[5]Legensky, S. M., Edwards, D. E., Bush, R. H., Poirier, D. M. A., Rumsey, C. L., Cosner, R. R., and Towne, C. E., "CFD General Notation System (CGNS): Status and Future Directions," AIAA Paper 2002-0752, 2002.

[6]Hauser, T., "Parallel I/O for the CGNS System," AIAA Paper 2004-1088, 2004.

[7]Poinot, M., Rumsey, C. L., and Mani, M., "Impact of CGNS on CFD Workflow," AIAA Paper 2004-2142, 2004.

[8]Hauser, T., "Benchmarking the CGNS I/O Performance," AIAA Paper 2008-0479, 2008.

[9]Hauser, T. and Horne, K., "CGNS I/O Performance on Parallel File Systems," AIAA Paper 2010-1057, 2010.

[10]Vassberg, J. C., Tinoco, E. N., Mani, M., Brodersen, O. P., Eisfeld, B., Wahls, R. A., Morrison, J. H., Zickuhr, T., Laflin, K. R., Mavriplis, D. J., "Abridged Summary of the Third AIAA Computational Fluid Dynamics Drag Prediction Workshop," *Journal of Aircraft*, Vol. 45, No. 3, 2008, pp. 781–798.

[11]Rumsey, C. L., Long, M., Stuever, R. A., and Wayman, T. R., "Summary of the First AIAA CFD High Lift Prediction Workshop (invited)," AIAA Paper 2011-939, 2011.

[12]Poinot, M., "Five Good Reasons to Use the Hierarchical Data Format," *Computing in Science & Engineering*, Scientific Programming, eds. Konstantin Laufer and Konrad Hinsen, Sept.-Oct. 2010, pp. 84–90.

[13]Pakalapati, P. D. and Hauser, T., "Benchmarking Parallel I/O Performance for Computational Fluid Dynamics Applications," AIAA Paper 2005-1381, 2005.

[14]Millman, J. and Vaught, T., "The State of SciPy," Proceedings of the 7th Python in Science Conference (SciPy 2008), G. Varoquaux, T. Vaught, J. Millman (Eds.), Pasadena, CA, 2009, pp. 5–10.

[15]Ortun, B., "CSM/CFD Coupling for the Dynamic Analysis of Helicopter Rotors," PhD Thesis, Conservatoire National des Arts et Metiers (CNAM), Paris, France, December 2008.

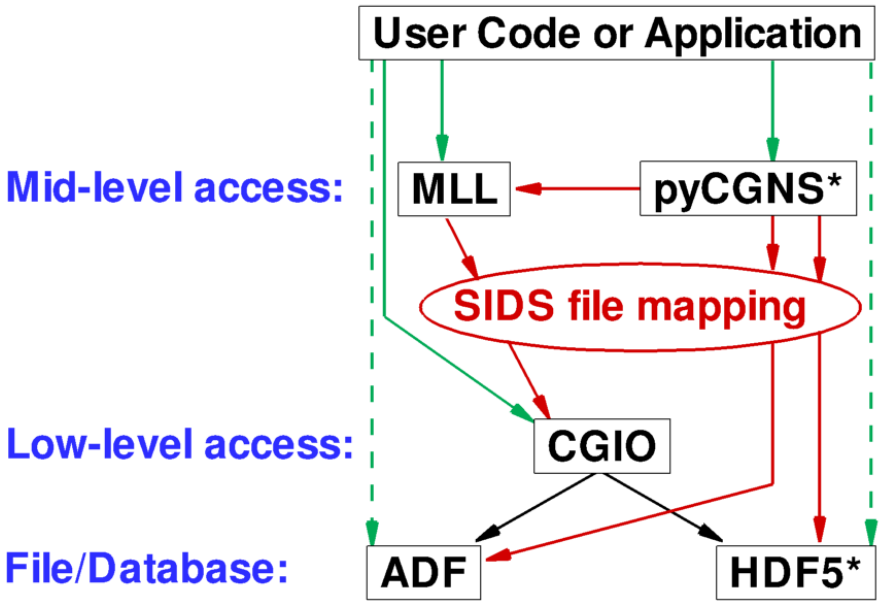


Figure 1. Overview of CGNS access levels to ADF and HDF5 files. Green lines indicate access from the user code or application, red lines indicate mid-level access calls (that make use of SIDS file mapping), and black lines indicate low-level access calls. *Note that HDF5 and pyCGNS (and related software) are not part of the standard CGNS software release, but are available separately.

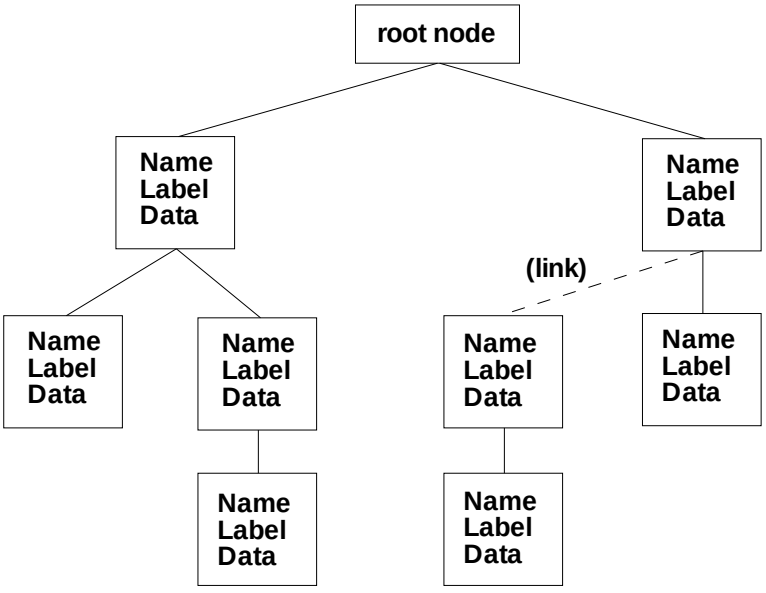
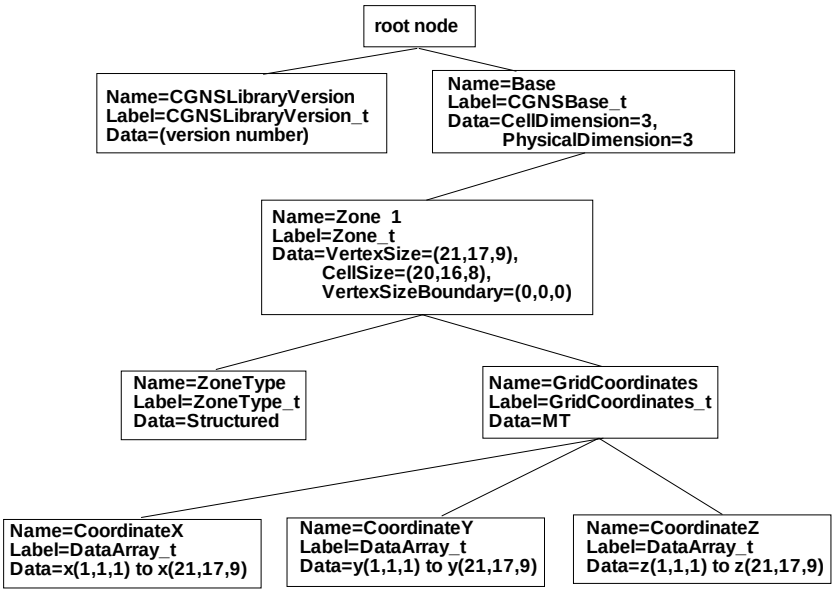
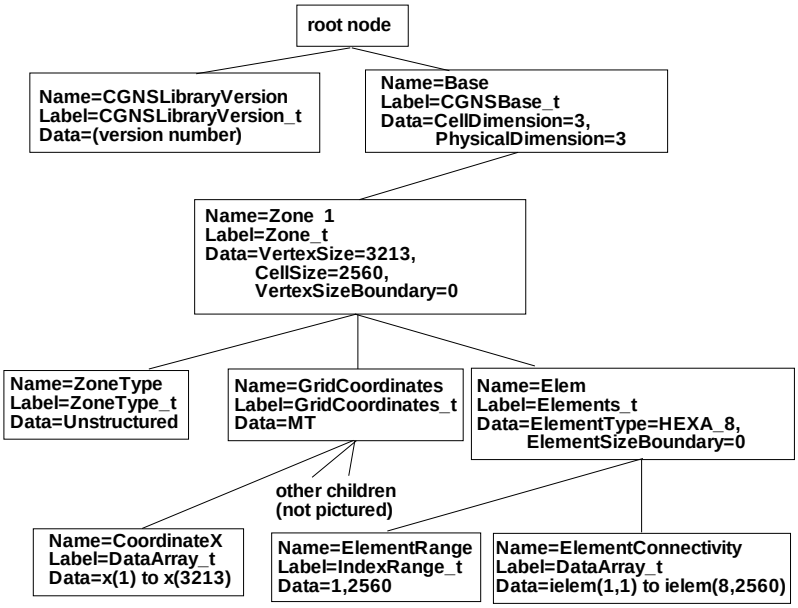


Figure 2. Example CGNS tree-like structure.



(a) **structured grid**



(b) **unstructured grid**

Figure 3. Examples showing layout of CGNS file structure.

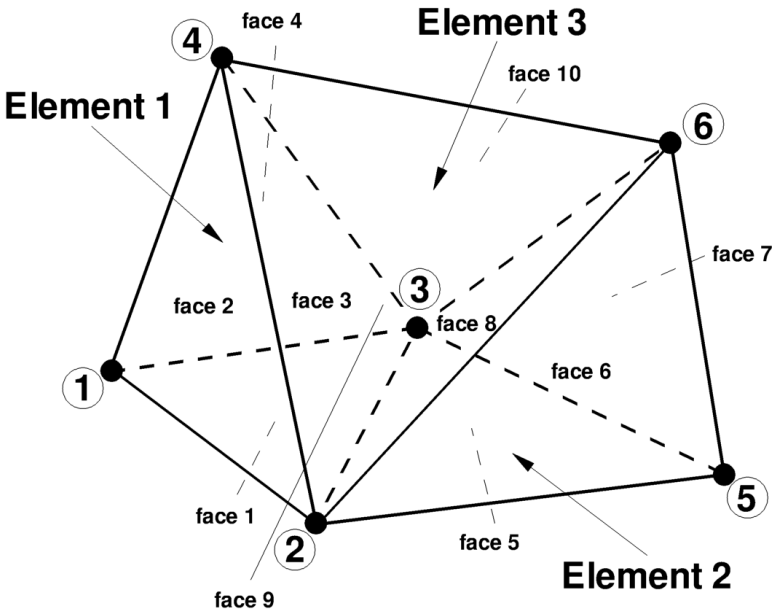


Figure 4. Simple example grid made up of 3 tetrahedra. Circled numbers indicate grid nodes.

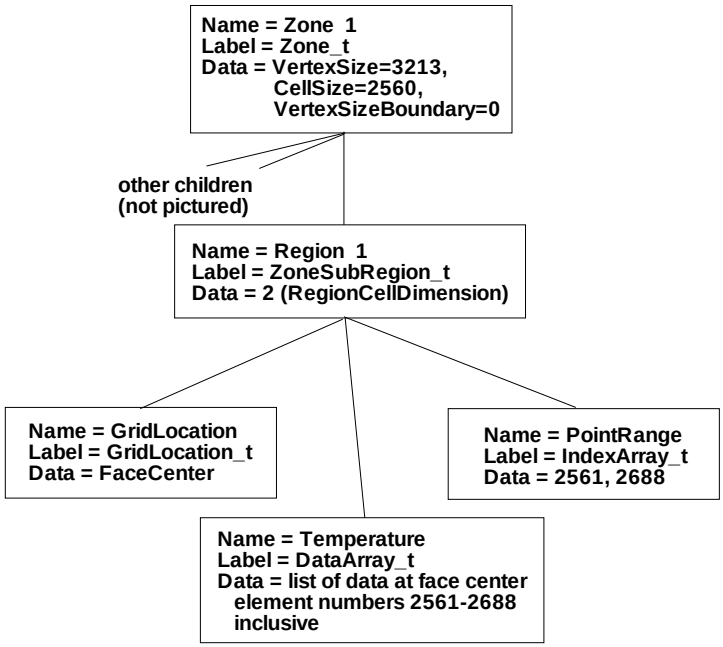


Figure 5. Example of typical Regions usage: surface subregion for a 3-D unstructured grid.

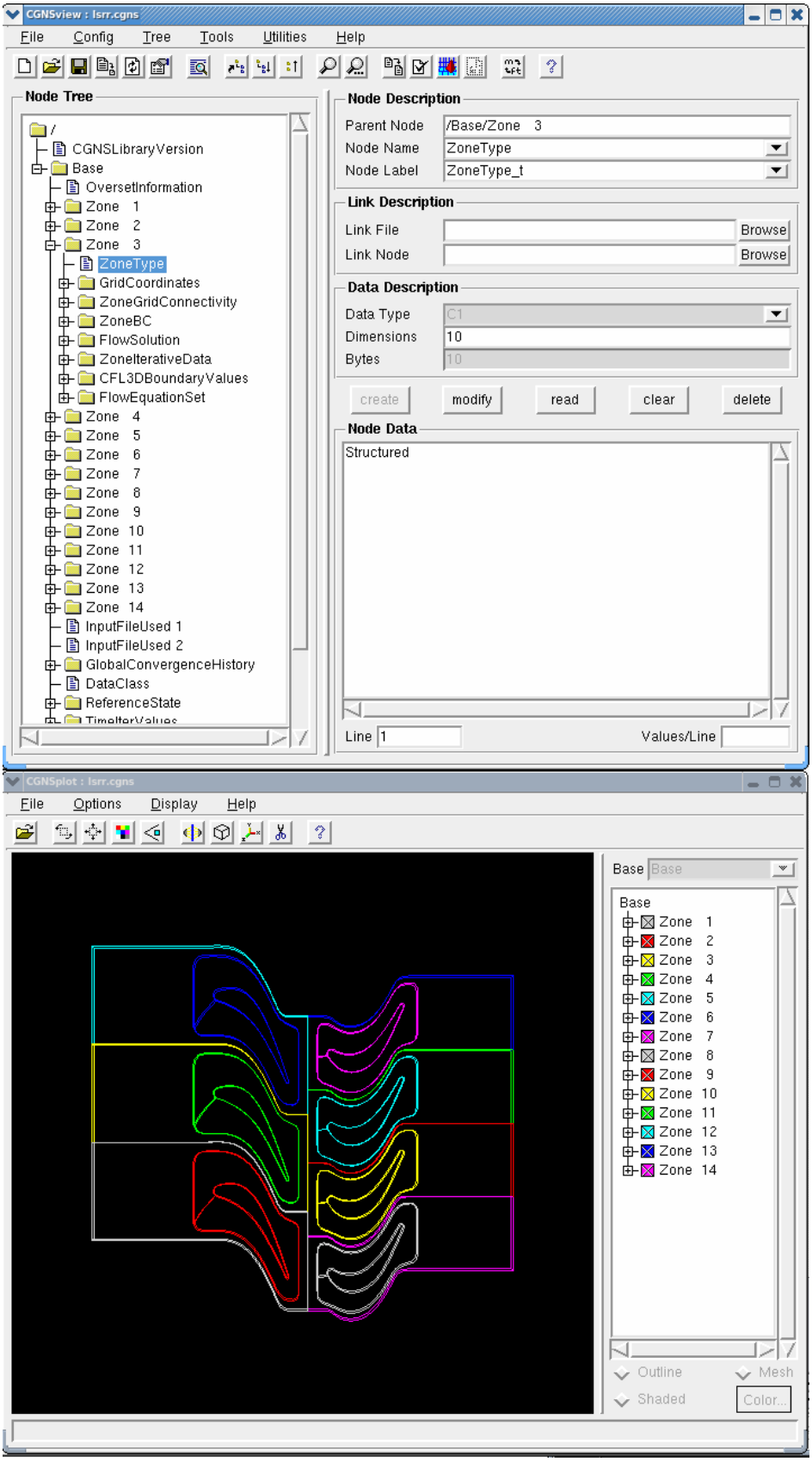


Figure 6. Example screen shot of CGNSview utility.