CFD General Notation System
A User's Guide To CGNS

Document Version 3.1.2

CGNS Version 3.1.3

Christopher L. Rumsey
NASA Langley Research Center

Diane M. A. Poirier
ICEM CFD Engineering

Robert H. Bush
Pratt & Whitney

Charles E. Towne
NASA Glenn Research Center

Contents

1 INTRODUCTION

This User's Guide (originally published as NASA/TM-2001-211236, October 2001) has been written to aid users in the implementation of CGNS (CFD General Notation System). It is intended as a tutorial: light in content, but heavy in examples, advice, and guidelines. Readers interested in additional details are referred to other documents, listed in the references, which are available from the CGNS website cgns.sourceforge.net.

1.1 What is CGNS?

CGNS (CFD General Notation System) 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, BCs, and auxiliary information. CGNS is also easily extensible, and allows for file-stamping and user-inserted-commenting. It employs ADF (Advanced Data Format), a system which creates binary files that are portable across computer platforms. CGNS also includes a second layer of software known as the mid-level library, or API (Application Programming Interface), which eases the implementation of CGNS into existing CFD codes.[1]

In 1999, control of CGNS was completely 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 completely free and open to anyone (open source). The CGNS standard is also the object of an ISO standardization effort for fluid dynamics data [6], for release some time in the early to mid-2000's.

1.2 Why CGNS?

CGNS will eventually eliminate most of the translator programs now necessary when working between machines and between CFD codes. Also, it eventually may allow for the results from one code to be easily restarted using another code. It will hopefully therefore save a lot of time and money. In particular, it is hoped that future grid-generation software will generate grids *with all connectivity and BC information included* as part of a CGNS database, saving time and avoiding potential costly errors in setting up this information after-the-fact.

[1] With the release of CGNS Version 2.4, the mid-level library may be built using either ADF or HDF (Hierarchical Data Format) as the underlying data format. The remainder of this *User's Guide* only refers to ADF, but at the mid-level library level the differences should be transparent to most users.

1.3 What is a CGNS File?

A CGNS file is an entity that is organized (inside the file itself) into a set of "nodes" in a tree-like structure, in much the same way as directories are organized in the UNIX environment. [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 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 a separate CGNS file altogether. Links are transparent to the user: the user "sees" linked children nodes as if they truly exist in the current tree. An example of a CGNS tree-like structure is shown in Fig. 1.

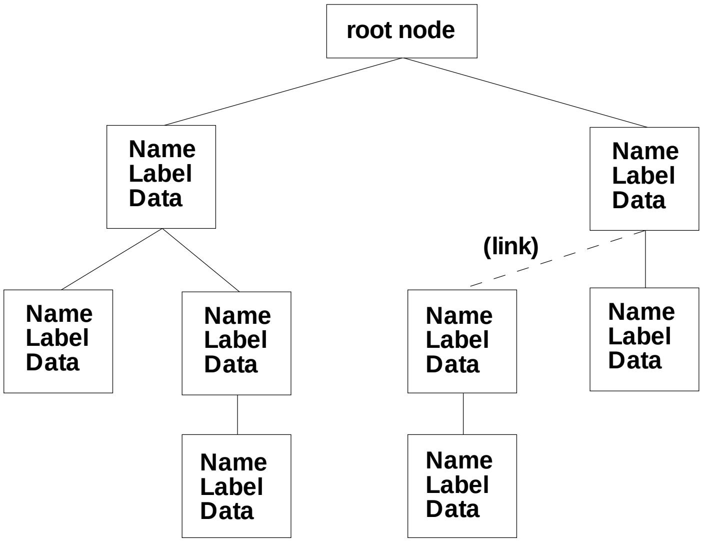

Figure 1: Example CGNS tree-like structure.

In order for any user to be able to interpret a CGNS file, its nodes must be assembled according to particular rules. For example, Fig. 2 shows a simple example of a tree-like structure that organizes some animals into categories according to rules that most of us are very familiar with. (Note that this figure is different from Fig. 1 in that no "Labels" or "Data" are used, only "Names.") The categories get narrower and narrower in their scope as you traverse lower in the tree. The broadest category here is "Animals," and the tree narrows all the way down to particular dogs (two "Fido"s, a "Spot," and a "Ginger"). Knowing ahead of time how this tree is organized allows you to quickly and easily access whatever particular information from the tree that you may be interested in. If someone else were to organize these same animals in a completely different way, according to different rules, then it would be difficult for you to access the desired information without spending a lot of time searching and studying the tree.

[2] Strictly speaking, because links may be used to store information in multiple files, there is no notion of a CGNS *file*, only of a CGNS *database* implemented within one or more files. However, throughout

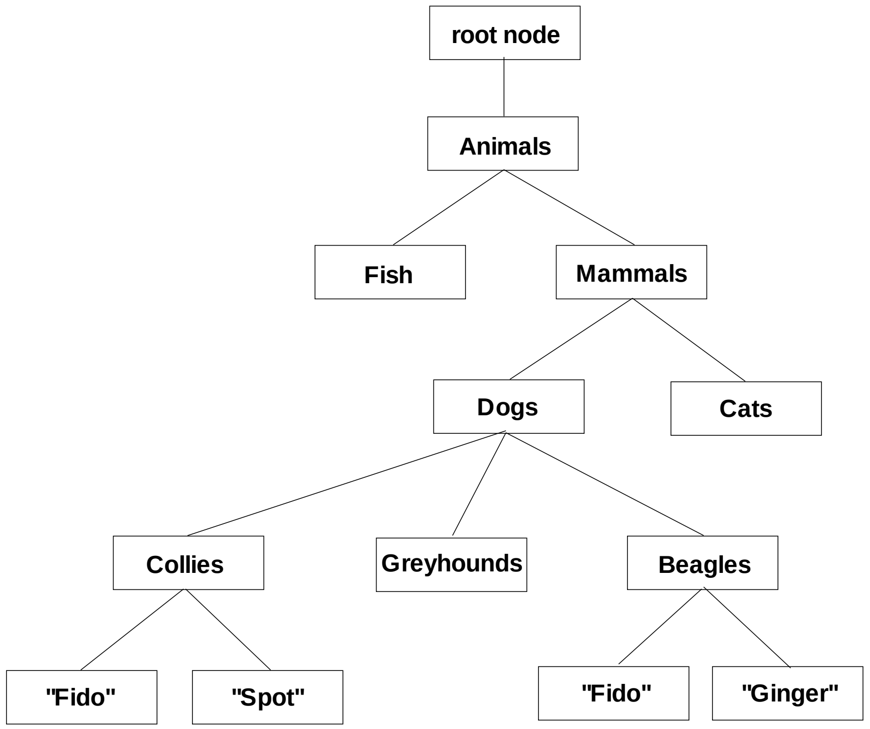

Figure 2: Simple tree-like structure that categorizes some animals.

The particular rules for organizing CGNS files for aerodynamic data, which allow users to easily access desired information, are described in the Standard Interface Data Structures (SIDS) document [1]. Because CGNS files are binary files, they cannot be viewed by the user with standard UNIX ASCII-editing tools. The utility CGNSview was created to allow users to easily view CGNS files.

1.4 How this User's Guide is Organized

The main content in this User's Guide is located in section 2, where several simple examples are given for both structured and unstructured grids. This section covers the basics that most users want or need to learn in order to get started using CGNS. It is recommended that the section on structured grids be read first, in its entirety, even if the user is only interested in unstructured grid applications. Some additional information is covered in section 3; these issues are felt to be important (i.e., most users will want to eventually include them), but they are not as crucial as the basic items covered in section 2. Finally, sections 4 and 5 briefly cover troubleshooting and frequently asked questions, respectively.

Note that all of the codes and code segments given in this document are available

this document the two phrases are used interchangeably.

as complete codes from the CGNS site at SourceForge (sourceforge.net/projects/cgns). The names of these codes and their functions are listed in Appendix A. Also note that not all CGNS capabilities are covered in this document. It is meant to be a fairly simple introductory guide only.

2 GETTING STARTED

The rules and conventions governing how the nodes in a CGNS file are organized, including their names and labels, are specified in the SIDS document [1], with additional details in [2] [3]. These documents also specify in detail how CFD information is to be stored within the nodes in a standardized fashion so that other users can easily access and read it. When a CGNS file strictly adheres to the rules given in the SIDS document, it is said to be "SIDS-compliant." A CGNS file must be SIDS-compliant in order for other users to be able to properly interpret it. A brief overview of the most commonly used aspects of the SIDS is given in Appendix B.

However, to get started with CGNS, it is not necessary for the user to fully understand the SIDS document or Appendix B. The mid-level, or API calls have been created to aid users in writing and reading CGNS files that are SIDS-compliant. [3] Using the API, most CFD data of interest to the majority of users can be written into or read from a CGNS file very easily with only an elementary understanding of the SIDS.

In the following sections, we give detailed instructions on how to create typical CGNS files or portions of files. These instructions are given in the form of simple examples. They make use of the mid-level API calls, although not all API calls are covered in this document (a complete list of available API calls can be found in [5]). We recommend that the user read through the examples in this section *in order*, because some information in the later sections depends on being familiar with information given in the earlier ones. Hopefully, users should be able to easily extend these simple examples to their own applications. Additional applications are covered in section 3. For those users already familiar with the PLOT3D format for CFD data [7], we include a detailed description on reading and writing PLOT3D-type variables in a CGNS file in Appendix C.

Also note that we have delayed the discussion of units and nondimensionalization until section 3. For now, all examples simply store and retrieve pure *numbers*, and it is assumed that the user knows what the dimensions or nondimensionalizations of each variable are.

[3] There are currently two levels of programming access to CGNS. The lowest level consists of ADF- or HDF5-level calls. These calls perform the most basic functions, such as creating a child node, writing data, reading data, etc. However, these low-level calls know nothing at all about the SIDS, so the user is responsible for putting data in the correct place, to make the CGNS file SIDS-compliant. The mid-level, or API calls, which always begin with the characters "cg_", were written with knowledge of the SIDS. Therefore, it is easier to adhere to the SIDS standards when writing a CGNS file using the API calls, and some checks for SIDS-compliance are also made by the API calls when accessing a CGNS file (SIDS compliance is not guaranteed, but the API calls go a long way toward facilitating it). The API calls also drastically shorten the calling sequences necessary to perform many of the functions needed to create and read CGNS files.

2.1 Structured Grid

This first section gives several structured grid examples, whereas section 2.2 gives unstructured grid examples. However, we recommend that section 2.1 be read first, in its entirety, even if the user is only interested in unstructured grid applications. This is because much of the organization of the CGNS files is identical for both grid types, and later sections of this document assume that the user is familiar with information given in earlier sections.

2.1.1 Single-Zone Structured Grid

This first example is for a very simple 3-D Cartesian grid of size 21 × 17 × 9. The grid points themselves are created using the following FORTRAN code snippet:

```
do k=1,nk
   do j=1,nj
      do i=1,ni
         x(i,j,k)=float(i-1)
         y(i,j,k)=float(j-1)
         z(i,j,k)=float(k-1)
      enddo
   enddo
enddo
```

where `ni=21`, `nj=17`, and `nk=9`. A picture of the grid is shown in Fig. 3.

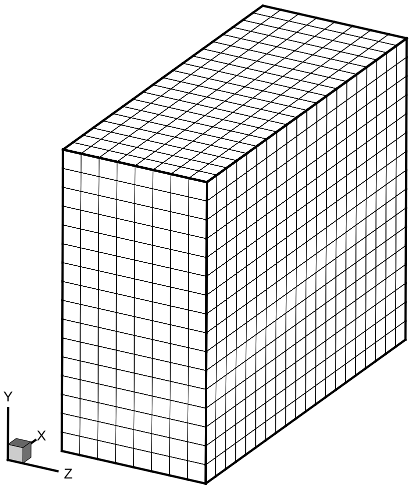

Figure 3: Simple Cartesian structured grid.

A complete FORTRAN code that creates this grid and uses API calls to write it to a CGNS file called `grid.cgns` is shown here (note that a FORTRAN line continuation is denoted by a `+`). This (and all later) coded examples are available from the CGNS site at SourceForge (sourceforge.net/projects/cgns). See Appendix A.

```
      program write_grid_str
c
c Creates simple 3-D structured grid and writes it to a
c CGNS file.
c
c This program uses the fortran convention that all
c variables beginning with the letters i-n are integers,
c by default, and all others are real
c
c Example compilation for this program is (change paths!):
c
c ifort -I ../CGNS_CVS/cgnslib -c write_grid_str.f
c ifort -o write_grid_str write_grid_str.o -L ../CGNS_CVS/cgnslib/LINUX -lcgns
c
c (../CGNS_CVS/cgnslib/LINUX/ is the location where the compiled
c library libcgns.a is located)
c
c cgnslib_f.h file must be located in directory specified by -I during compile:
      include 'cgnslib_f.h'
c dimension statements (note that tri-dimensional arrays
c x,y,z must be dimensioned exactly as (21,17,N) (N>=9)
c for this particular case or else they will be written to
c the CGNS file incorrectly!  Other options are to use 1-D
c arrays, use dynamic memory, or pass index values to a
c subroutine and dimension exactly there):
      real*8 x(21,17,9),y(21,17,9),z(21,17,9)
      dimension isize(3,3)
      character basename*32,zonename*32
c
c create gridpoints for simple example:
      ni=21
      nj=17
      nk=9
      do k=1,nk
         do j=1,nj
            do i=1,ni
               x(i,j,k)=float(i-1)
               y(i,j,k)=float(j-1)
               z(i,j,k)=float(k-1)
            enddo
         enddo
      enddo
      write(6,'('' created simple 3-D grid points'')')
c
c WRITE X, Y, Z GRID POINTS TO CGNS FILE
c open CGNS file for write
      call cg_open_f('grid.cgns',CG_MODE_WRITE,index_file,ier)
      if (ier .ne.  CG_OK) call cg_error_exit_f
c create base (user can give any name)
      basename='Base'
      icelldim=3
      iphysdim=3
      call cg_base_write_f(index_file,basename,icelldim,iphysdim,
+    index_base,ier)
c define zone name (user can give any name)
      zonename = 'Zone  1'
c vertex size
      isize(1,1)=21
      isize(2,1)=17
      isize(3,1)=9
c cell size
      isize(1,2)=isize(1,1)-1
      isize(2,2)=isize(2,1)-1
      isize(3,2)=isize(3,1)-1
c boundary vertex size (always zero for structured grids)
      isize(1,3)=0
      isize(2,3)=0
      isize(3,3)=0
c create zone
      call cg_zone_write_f(index_file,index_base,zonename,isize,
+    Structured,index_zone,ier)
c write grid coordinates (user must use SIDS-standard names here)
      call cg_coord_write_f(index_file,index_base,index_zone,RealDouble,
+    'CoordinateX',x,index_coord,ier)
      call cg_coord_write_f(index_file,index_base,index_zone,RealDouble,
+    'CoordinateY',y,index_coord,ier)
      call cg_coord_write_f(index_file,index_base,index_zone,RealDouble,
+    'CoordinateZ',z,index_coord,ier)
c close CGNS file
      call cg_close_f(index_file,ier)
      write(6,'('' Successfully wrote grid to file grid.cgns'')')
      stop
      end
```

There are several items to note regarding this code. Whenever a new entity is created using the API, an integer index is returned. This index is used in subsequent API calls to refer to the entity. For example, the above call to `cg_open_f`, which opens the file grid.cgns, assigns to this entity the index `index_file`. This same `index_file` is used to identify this entity in subsequent calls. Similarly, `cg_base_write_f` assigns an index `index_base` to the base, `cg_zone_write_f` assigns an index `index_zone` to the zone, and `cg_coord_write_f` assigns an index `index_coord` to each coordinate.

For FORTRAN code, an include statement pointing to `cgnslib_f.h` must be present. (The cgnslib_f.h file comes with the CGNS software.) Also, it is imperative that the `x`, `y`, and `z` arrays be dimensioned *exactly* as (21,17,N), where $N \geq 9$ (or else as a one-dimensional array of at least size $21 * 17 * 9$) for this particular example; this is because the `cg_coord_write_f` routine writes the first $21 * 17 * 9$ values contained in the array *as it is stored in memory*. If `x`, `y`, and `z` are tri-dimensional arrays and the first two indices are dimensioned larger than 21 and 17, respectively, then incorrect values will be placed in the CGNS file. In a real working code, one would probably either (a) use one-dimensional arrays, (b) dynamically allocate appropriate memory for `x`, `y`, and `z`, or else (c) pass the index values to a subroutine and write via an appropriately dimensioned work array.

In this case, the cell dimension (`icelldim`) is 3 (because the grid is made up of volume cells), and the physical dimension (`iphysdim`) is 3 (because 3 coordinates define 3-D). (Refer to Appendix B for a more detailed description.) The `isize` array contains the vertex size, cell size, and boundary vertex size for each index direction. For a 3-D structured grid, the index dimension is always the same as the cell dimension, so this means there are 3 vertex sizes, 3 cell sizes, and 3 boundary vertex sizes (one each for the i, j, and k directions). For structured grids, the cell size is always one less than the corresponding vertex size, and the boundary vertex size has no meaning and is always zero. When writing the grid coordinates, the user must use SIDS-standard names. For example, `x`, `y`, and `z` coordinates must be named `CoordinateX`, `CoordinateY`, and `CoordinateZ`, respectively. Other standard names exist for other possible choices (see [1]). Finally, `basename` and `zonename` must be declared as character strings, and the integer array `isize` must be dimensioned appropriately.

The grid coordinate arrays can be written in single or double precision. The desired data type is communicated to the API using the keywords `RealSingle` or `RealDouble`. The user must insure that the data type transmitted to the API is consistent with the the one used in declaring the coordinates arrays. When it is compiled, the code must also link to the compiled CGNS library `libcgns.a`. Instructions for compiling the CGNS library are given in README files that come with the CGNS software.

A complete code written in C that performs the same task of creating grid coordinates and writing them to a CGNS file is given here.

```
/* Program write_grid_str.c */
```

```
/*
Creates simple 3-D structured grid and writes it to a
CGNS file.

Example compilation for this program is (change paths!):

cc -I ../CGNS_CVS/cgnslib -c write_grid_str.c
cc -o write_grid_str_c write_grid_str.o -L ../CGNS_CVS/cgnslib/LINUX -lcgns

(../CGNS_CVS/cgnslib/LINUX/ is the location where the compiled library libcgns.a
is located)
*/

#include <stdio.h>
#include <string.h>
/* cgnslib.h file must be located in directory specified by -I during compile:
*/
#include "cgnslib.h"

#if CGNS_VERSION < 3100
# define cgsize_t int
#else
# if CG_BUILD_SCOPE
# error enumeration scoping needs to be off
# endif
#endif

int main()
{
/*
   dimension statements (note that tri-dimensional arrays
   x,y,z must be dimensioned exactly as [N][17][21] (N>=9)
   for this particular case or else they will be written to
   the CGNS file incorrectly!  Other options are to use 1-D
   arrays, use dynamic memory, or pass index values to a
   subroutine and dimension exactly there):
*/
   double x[9][17][21],y[9][17][21],z[9][17][21];
   cgsize_t isize[3][3];
   int ni,nj,nk,i,j,k;
   int index_file,icelldim,iphysdim,index_base;
   int index_zone,index_coord;
   char basename[33],zonename[33];

/* create gridpoints for simple example:  */
   ni=21;
   nj=17;
   nk=9;
   for (k=0; k < nk; ++k)
   {
      for (j=0; j < nj; ++j)
      {
         for (i=0; i < ni; ++i)
         {
            x[k][j][i]=i;
            y[k][j][i]=j;
            z[k][j][i]=k;
         }
      }
   }
   printf("\ncreated simple 3-D grid points");
/* WRITE X, Y, Z GRID POINTS TO CGNS FILE */
/* open CGNS file for write */
   if (cg_open("grid_c.cgns",CG_MODE_WRITE,&index_file)) cg_error_exit();
/* create base (user can give any name) */
   strcpy(basename,"Base");
   icelldim=3;
   iphysdim=3;
   cg_base_write(index_file,basename,icelldim,iphysdim,&index_base);
/* define zone name (user can give any name) */
   strcpy(zonename,"Zone 1");
/* vertex size */
   isize[0][0]=21;
   isize[0][1]=17;
   isize[0][2]=9;
/* cell size */
   isize[1][0]=isize[0][0]-1;
   isize[1][1]=isize[0][1]-1;
   isize[1][2]=isize[0][2]-1;
/* boundary vertex size (always zero for structured grids) */
   isize[2][0]=0;
   isize[2][1]=0;
   isize[2][2]=0;
/* create zone */
   cg_zone_write(index_file,index_base,zonename,*isize,Structured,
      &index_zone);
/* write grid coordinates (user must use SIDS-standard names here) */
   cg_coord_write(index_file,index_base,index_zone,RealDouble,"CoordinateX",
      x,&index_coord);
   cg_coord_write(index_file,index_base,index_zone,RealDouble,"CoordinateY",
      y,&index_coord);
   cg_coord_write(index_file,index_base,index_zone,RealDouble,"CoordinateZ",
      z,&index_coord);
```

```
/* close CGNS file */
   cg_close(index_file);
   printf("\nSuccessfully wrote grid to file grid_c.cgns\n");
   return 0;
}
```

Note that in the C-code, the ".h" file that must be included is called `cgnslib.h`. From now on, all codes will be given in FORTRAN only. The C-equivalent calls are similar, as demonstrated above. Also, from now on, complete code will not be shown, but rather only code segments, in order to save space. However, complete codes can be accessed from the CGNS site at SourceForge (sourceforge.net/projects/cgns).

The CGNS file `grid.cgns` that is created by the code above is a binary file that, internally, possesses the tree-like structure shown in Fig. 4. As mentioned in the Introduction, each node has a name, a label, and may or may not contain data. In the example in the figure, all the nodes contain data except for the `GridCoordinates` node, for which `MT` indicates no data.

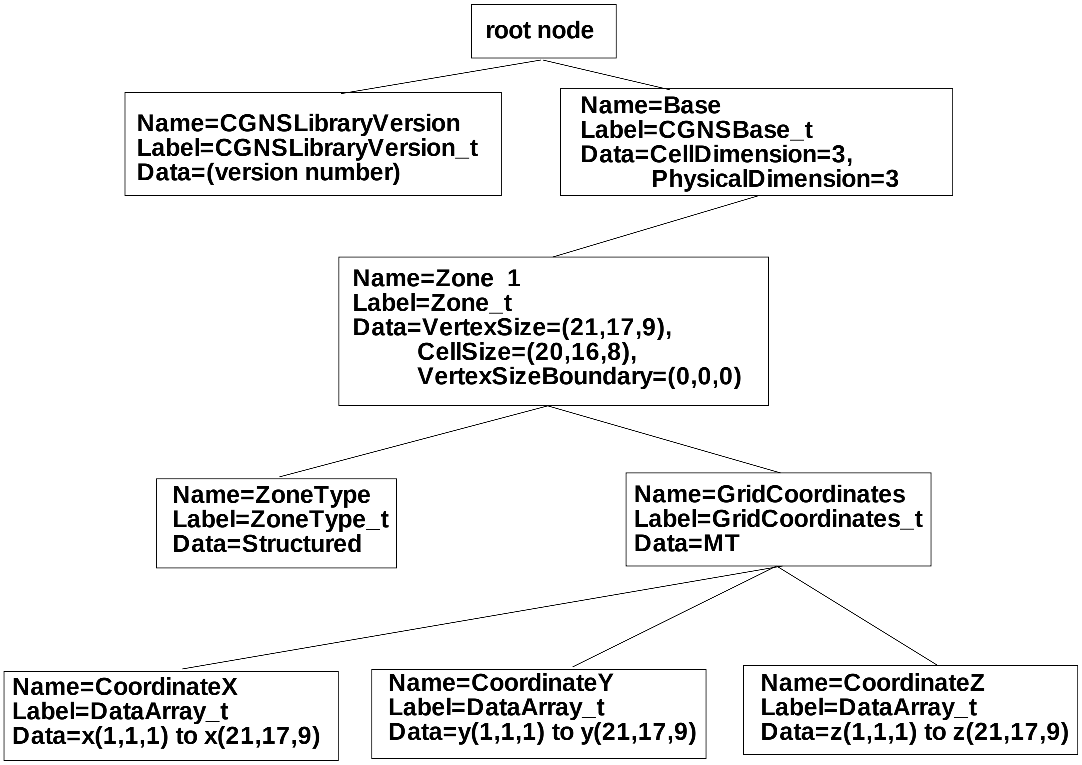

Figure 4: Layout of CGNS file for simple Cartesian structured grid.

However, the user really does not need to know the full details of the tree-like structure in this case. The API has automatically created a SIDS-compliant CGNS file! Now, the user can just as easily read the CGNS file using the API. The FORTRAN code segment used to read the CGNS file `grid.cgns` that we just created is given here:

```
c READ X, Y, Z GRID POINTS FROM CGNS FILE
   include 'cgnslib_f.h'
c open CGNS file for read
   call cg_open_f('grid.cgns',CG_MODE_READ,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
c we know there is only one zone (real working code would check!)
   index_zone=1
c get zone size (and name - although not needed here)
   call cg_zone_read_f(index_file,index_base,index_zone,zonename,
+    isize,ier)
c lower range index
   irmin(1)=1
   irmin(2)=1
   irmin(3)=1
c upper range index of vertices
   irmax(1)=isize(1,1)
   irmax(2)=isize(2,1)
   irmax(3)=isize(3,1)
c read grid coordinates
   call cg_coord_read_f(index_file,index_base,index_zone,
+    'CoordinateX',RealSingle,irmin,irmax,x,ier)
   call cg_coord_read_f(index_file,index_base,index_zone,
+    'CoordinateY',RealSingle,irmin,irmax,y,ier)
   call cg_coord_read_f(index_file,index_base,index_zone,
+    'CoordinateZ',RealSingle,irmin,irmax,z,ier)
c close CGNS file
   call cg_close_f(index_file,ier)
```

Note that this FORTRAN coding is very rudimentary. It assumes that we know that there is only one base and one zone. In a real working code, one should check the numbers in the file, and either allow for the possibility of multiple bases or zones, or explicitly disallow it. Also, this coding implicitly assumes that the `grid.cgns` file is a 3-D structured grid (cell dimension = physical dimension = 3). In a real working code, one should check to make sure that this is true, or else allow for other possibilities. One should also check to make sure the zone type is `Structured` if this is the type expected.

As before, the `x`, `y`, and `z` arrays in this case *must* be dimensioned correctly: for a tri-dimensional array, (21,17,N), where $N \geq 9$. (In a real working code, one would probably either (a) use one-dimensional arrays, (b) dynamically allocate appropriate memory for `x`, y, and z after reading isize, or else (c) pass the isize values to a subroutine and dimension a work array appropriately prior to reading.) Also note that, regardless of the precision in which the grid coordinates were written to the CGNS file (single or double), one can read them either way; the API automatically performs the translation. (The arrays x, y, and z in the code above must be declared as single precision if RealSingle is used and as double precision if RealDouble is used.) Finally, isize should be dimensioned appropriately, zonename should be declared as a character variable, and irmin and irmax should be dimensioned appropriately.

2.1.2 Single-Zone Structured Grid and Flow Solution

In this section, we now write a flow solution associated with the grid from section 2.1.1. We assume that we have two flow solution arrays available: static density and static pressure. To illustrate three important options, we will show how to write the flow solution (a) at vertices, (b) at cell centers, and (c) at cell centers plus rind cells.

(a) Flow Solution at Vertices

The first option is illustrated schematically in 2-D in Fig. 5. Simply stated, a Vertex flow solution is located at the same location as the grid points. Assuming that the grid points have already been written to a CGNS file, the following FORTRAN code segment adds the flow solution at vertices:

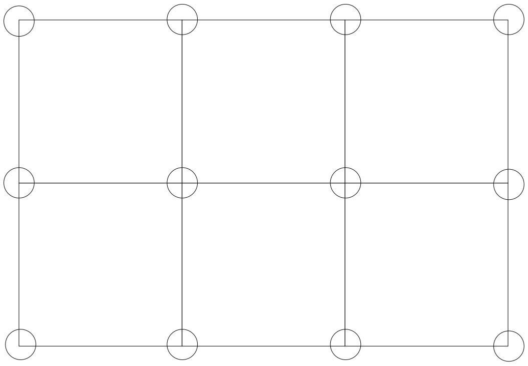

Figure 5: Schematic showing location (circles) of Vertex flow solution relative to grid.

```
c WRITE FLOW SOLUTION TO EXISTING CGNS FILE
   include 'cgnslib_f.h'
c open CGNS file for modify
   call cg_open_f('grid.cgns',CG_MODE_MODIFY,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
c we know there is only one zone (real working code would check!)
   index_zone=1
c define flow solution node name (user can give any name)
   solname = 'FlowSolution'
c create flow solution node
   call cg_sol_write_f(index_file,index_base,index_zone,solname,
+    Vertex,index_flow,ier)
c write flow solution (user must use SIDS-standard names here)
   call cg_field_write_f(index_file,index_base,index_zone,index_flow,
+    RealDouble,'Density',r,index_field,ier)
   call cg_field_write_f(index_file,index_base,index_zone,index_flow,
+    RealDouble,'Pressure',p,index_field,ier)
c close CGNS file
   call cg_close_f(index_file,ier)
```

In this code, the density (`r`) and pressure (`p`) variables *must* be dimensioned correctly for this particular case: for a tri-dimensional array, (21,17,N), where $N \geq 9$ (see discussion in section 2.1.1). Note that the API, knowing that the flow solution type is `Vertex`, automatically writes out the correct index range, corresponding with the zone's grid index range. Also note that we opened the existing CGNS file and modified it (`CG_MODE_MODIFY`) - we knew ahead of time that only one base and only one zone exist; a real working code would make appropriate checks. Finally, `solname` should be declared as a character variable and `r` and `p` must be declared as double precision variables when `RealDouble` type is used.

The layout of the CGNS file with the flow solution at vertices included is shown in Fig. 6. The three nodes under `GridCoordinates_t` have been left out to conserve space in the figure, but they exist as indicated by the three unconnected lines.

The vertex flow solution can be read in using the following FORTRAN code segment (can read in as single or double precision – see discussion in section 2.1.1):

```
c READ FLOW SOLUTION FROM CGNS FILE
   include 'cgnslib_f.h'
c open CGNS file for read
   call cg_open_f('grid.cgns',CG_MODE_READ,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
c we know there is only one zone (real working code would check!)
   index_zone=1
c we know there is only one FlowSolution_t (real working code would check!)
```

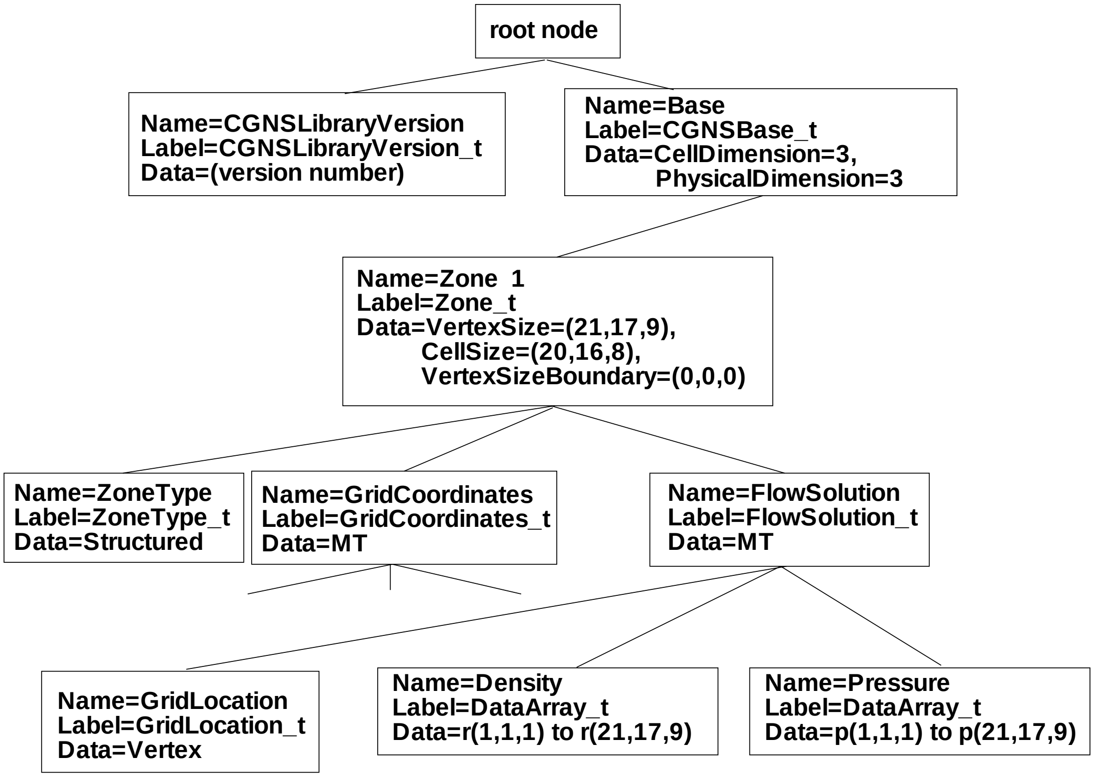

Figure 6: Layout of CGNS file for simple Cartesian structured grid with flow solution at vertices. (Note: because `GridLocation` = `Vertex` is the default, it is not necessary to specify it.)

```
   index_flow=1
c get zone size (and name - although not needed here)
   call cg_zone_read_f(index_file,index_base,index_zone,zonename,
+    isize,ier)
c lower range index
   irmin(1)=1
   irmin(2)=1
   irmin(3)=1
c upper range index - use vertex dimensions
c checking GridLocation first (real working code would check
c to make sure there are no Rind cells also!):
   call cg_sol_info_f(index_file,index_base,index_zone,index_flow,
+    solname,loc,ier)
   if (loc .ne.  Vertex) then
      write(6,'('' Error, GridLocation must be Vertex!'')')
      stop
   end if
   irmax(1)=isize(1,1)
   irmax(2)=isize(2,1)
   irmax(3)=isize(3,1)
c read flow solution
   call cg_field_read_f(index_file,index_base,index_zone,index_flow,
+    'Density',RealSingle,irmin,irmax,r,ier)
   call cg_field_read_f(index_file,index_base,index_zone,index_flow,
+    'Pressure',RealSingle,irmin,irmax,p,ier)
c close CGNS file
   call cg_close_f(index_file,ier)
```

Note that this code segment assumes that it is known that the flow solution contains no rind data (to be covered in detail below). If rind data *does* exist, but the user does not account for it, then the flow solution information will be read incorrectly. Hence, a real working code would check for rind cells, and adjust the dimensions and index ranges appropriately. Other similar cautions as those mentioned earlier regarding dimensioning of variables, real working code checks, etc., apply here as well. These cautions will not always be repeated from this point forward.

(b) Flow Solution at Cell Centers

The option for outputting the flow solution at cell centers is illustrated schematically in 2-D in Fig. 7. The flow solutions are defined at the *centers* of the cells defined by the four surrounding grid points. In 3-D, the cell centers are defined by eight surrounding grid points. The code segment to write to cell centers is identical to that given above for vertices, except that the call to `cg_sol_write_f` is replaced by:

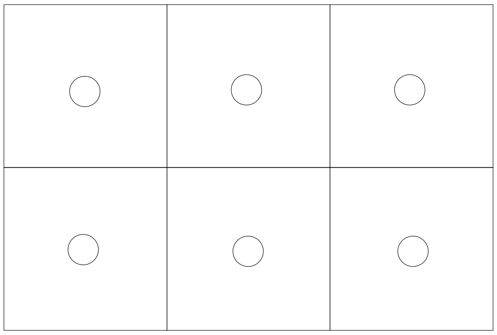

Figure 7: Schematic showing location (circles) of `CellCenter` flow solution relative to grid.

```
c create flow solution node (NOTE USE OF CellCenter HERE)
   call cg_sol_write_f(index_file,index_base,index_zone,solname,CellCenter,
+    index_flow,ier)
```

Also, now the density (`r`) and pressure (`p`) variables must be dimensioned correctly for this particular case: for a tri-dimensional array, (20,16,N), where $N \geq 8$ (i.e., one less in each index dimension than the grid itself). Again, the API, knowing that the flow solution type is `CellCenter`, automatically writes out the correct index range, corresponding with the zone's grid index range minus 1 in each index direction.

The layout of the CGNS file with the flow solution at cell centers is shown (below the `FlowSolution_t` node only) in Fig. 8. Note that the indices over which the flow solutions are written are now from $(1, 1, 1)$ to $(20, 16, 8)$ (contrast with the `FlowSolution` part of Fig. 6).

The FORTRAN code segment to read in the solution at cell centers is the same as that given above for vertices, except that the section that defines `irmax` is replaced by:

```
c upper range index - use cell dimensions
c checking GridLocation first (real working code would check
c to make sure there are no Rind cells also!):
   call cg_sol_info_f(index_file,index_base,index_zone,index_flow,
+    solname,loc,ier)
   if (loc .ne.  CellCenter) then
      write(6,'('' Error, GridLocation must be CellCenter!'')')
      stop
   end if
```

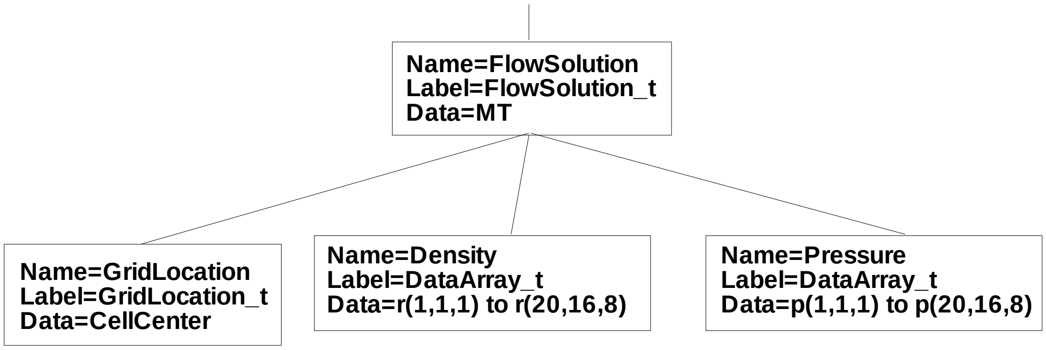

Figure 8: Layout of CGNS file (under `FlowSolution_t` node) for simple Cartesian structured grid with flow solution at cell centers.

```
irmax(1)=isize(1,2)
irmax(2)=isize(2,2)
irmax(3)=isize(3,2)
```

and, as usual, the `r` and `p` arrays must be dimensioned appropriately.

(c) Flow Solution at Cell Centers With Additional Rind Data

Rind data is additional flow solution data *exterior* to a grid, at "ghost" locations. Rind data can be associated with other `GridLocation` values beside `CellCenter`, although we only show an example using `CellCenter` here. Furthermore, this example is for structured grids only, for which Rind data can be defined implicitly (via indexing conventions alone). The option for outputting the flow solution at cell centers with additional rind data is illustrated schematically in 2-D in Fig. 9. In this diagram, we show one layer of rind cell data in the row below the grid itself. There could be rind data at other sides of the grid, or there could be more than one row at a given side.

In CGNS, the flow solution at rind cells is not stored as separate entities, but rather the flow solution range is extended to *include* the rind cells. For example, in the 2-D schematic of Fig. 9, instead of an index range of `p(3,2)` for pressures stored at the cell centers, the flow solution would now have an index range of `p(3,0:2)` or `p(3,3)`. See [1] for details.

For our 3-D example, we assume that we have one row of rind data at 4 faces of the zone (`ilo, ihi, jlo, jhi`, where these represent the low and high ends of the i and j directions, respectively), and no rind cells at `klo` or `khi` (at either end of the k direction). The code segment to write the flow solution and rind data is as follows:

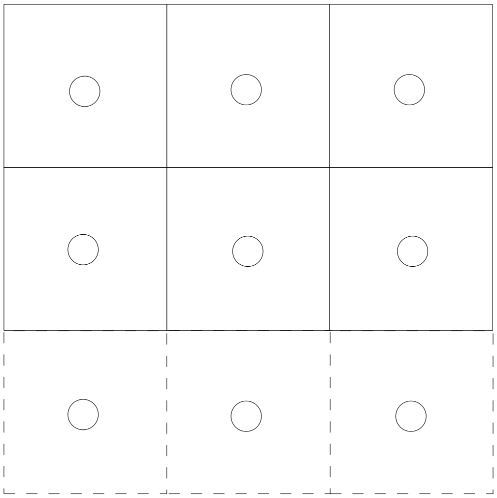

Figure 9: Schematic showing location (circles) of `CellCenter` flow solution, including rind cells, relative to grid.

```
c WRITE FLOW SOLUTION TO EXISTING CGNS FILE
   include 'cgnslib_f.h'
c open CGNS file for modify
   call cg_open_f('grid.cgns',CG_MODE_MODIFY,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
c we know there is only one zone (real working code would check!)
   index_zone=1
c define flow solution node name (user can give any name)
   solname = 'FlowSolution'
c create flow solution node
   call cg_sol_write_f(index_file,index_base,index_zone,solname,CellCenter,
+    index_flow,ier)
c go to position within tree at FlowSolution_t node
   call cg_goto_f(index_file,index_base,ier,'Zone_t',index_zone,
+    'FlowSolution_t',index_flow,'end')
c write rind information under FlowSolution_t node (ilo,ihi,jlo,jhi,klo,khi)
   irinddata(1)=1
   irinddata(2)=1
   irinddata(3)=1
   irinddata(4)=1
   irinddata(5)=0
   irinddata(6)=0
   call cg_rind_write_f(irinddata,ier)
c write flow solution (user must use SIDS-standard names here)
```

```
   call cg_field_write_f(index_file,index_base,index_zone,index_flow,
+    RealDouble,'Density',r,index_field,ier)
   call cg_field_write_f(index_file,index_base,index_zone,index_flow,
+    RealDouble,'Pressure',p,index_field,ier)
c close CGNS file
   call cg_close_f(index_file,ier)
```

Note that in the case of rind data, the user must position the `Rind_t` node appropriately, using the `cg_goto_f` call. In this case, the `Rind_t` node belongs under the `FlowSolution_t` node.

For this case of cell center flow solution with rind data, the density (`r`) and pressure (`p`) are written to the CGNS file with the following index ranges: from $i = 0$ to $i = 20{+}1 = 21$ (or a total i length of 22), from $j = 0$ to $j = 16 + 1 = 17$ (or a total j length of 18), and from $k = 1$ to $k = 8$. The variables `r` and `p` must be dimensioned appropriately to reflect these index ranges modified by the rind values.

The layout of the CGNS file for this example (below the `FlowSolution_t` node only) is shown in Fig. 10. Compare this figure with Figs. 6 and 8.

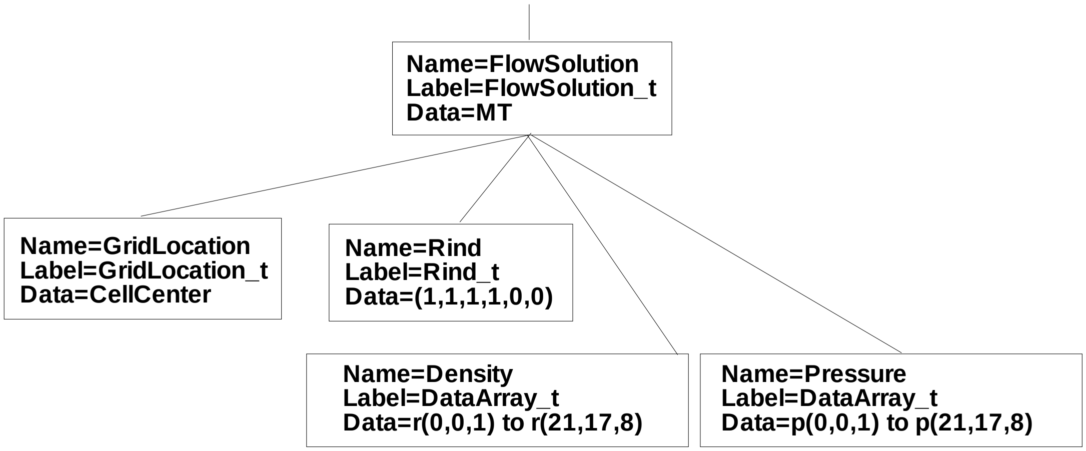

Figure 10: Layout of CGNS file (under `FlowSolution_t` node) for simple Cartesian structured grid with flow solution at cell centers plus rind data.

A FORTRAN code segment to read the flow solution for this example is:

```
c READ FLOW SOLUTION FROM CGNS FILE
   include 'cgnslib_f.h'
c open CGNS file for read
   call cg_open_f('grid.cgns',CG_MODE_READ,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
```

```
   index_base=1
c we know there is only one zone (real working code would check!)
   index_zone=1
c we know there is only one FlowSolution_t (real working code would check!)
   index_flow=1
c get zone size (and name - although not needed here)
   call cg_zone_read_f(index_file,index_base,index_zone,zonename,isize,ier)
c go to position within tree at FlowSolution_t node
   call cg_goto_f(index_file,index_base,ier,'Zone_t',index_zone,
+    'FlowSolution_t',index_flow,'end')
c read rind data
   call cg_rind_read_f(irinddata,ier)
c lower range index
   irmin(1)=1
   irmin(2)=1
   irmin(3)=1
c upper range index - use cell dimensions and rind info
c checking GridLocation first:
   call cg_sol_info_f(index_file,index_base,index_zone,index_flow,
+    + solname,loc,ier)
   if (loc .ne.  CellCenter) then
      write(6,'('' Error, GridLocation must be CellCenter!'')')
      stop
   end if
   irmax(1)=isize(1,2)+irinddata(1)+irinddata(2)
   irmax(2)=isize(2,2)+irinddata(3)+irinddata(4)
   irmax(3)=isize(3,2)+irinddata(5)+irinddata(6)
c read flow solution
   call cg_field_read_f(index_file,index_base,index_zone,index_flow,
+    'Density',RealSingle,irmin,irmax,r,ier)
   call cg_field_read_f(index_file,index_base,index_zone,index_flow,
+    'Pressure',RealSingle,irmin,irmax,p,ier)
c close CGNS file
   call cg_close_f(index_file,ier)
```

2.1.3 Single-Zone Structured Grid with Boundary Conditions

To illustrate the use of boundary conditions, we again use the same single-zone Cartesian grid from section 2.1.1. Referring back to Fig. 3, we wish to apply the following:

`ilo` – `BCTunnelInflow`
`ihi` – `BCExtrapolate`
`jlo` – `BCWallInviscid`
`jhi` – etc.

klo – etc.
khi – etc.

where BCTunnelInflow, BCExtrapolate, and BCWallInviscid are data-name identifiers for boundary conditions. The complete list of boundary condition identifiers is found in [1]. In this example, we take the approach of using the lowest-level BC implementation allowed – see Fig. 24 and the discussion in Appendix B.

In this section, we show two different approaches for defining the region over which each boundary condition acts. The first is with type PointRange, meaning that we define the minimum and maximum points on a face that define a logically rectangular region (this method is usable only for faces that are capable of being defined in this way). The second is with type PointList, which gives the list of *all* the points for which the boundary condition applies. This latter method is generally used for any zone whose defined region is not logically rectangular.

(a) Boundary Conditions Specifying Range

A FORTRAN code segment to write the boundary condition information of type PointRange to the existing CGNS file from section 2.1.1 or 2.1.2 is given here:

```
c WRITE BOUNDARY CONDITIONS TO EXISTING CGNS FILE
   include 'cgnslib_f.h'
c open CGNS file for modify
   call cg_open_f('grid.cgns',CG_MODE_MODIFY,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
c we know there is only one zone (real working code would check!)
   index_zone=1
c get zone size (and name - although not needed here)
   call cg_zone_read_f(index_file,index_base,index_zone,zonename,
+    isize,ier)
   ilo=1
   ihi=isize(1,1)
   jlo=1
   jhi=isize(2,1)
   klo=1
   khi=isize(3,1)
c write boundary conditions for ilo face, defining range first
c (user can give any name)
c lower point of range
   ipnts(1,1)=ilo
   ipnts(2,1)=jlo
   ipnts(3,1)=klo
```

```
c upper point of range
   ipnts(1,2)=ilo
   ipnts(2,2)=jhi
   ipnts(3,2)=khi
   call cg_boco_write_f(index_file,index_base,index_zone,'Ilo',
+    BCTunnelInflow,PointRange,2,ipnts,index_bc,ier)
c write boundary conditions for ihi face, defining range first
c (user can give any name)
c lower point of range
   ipnts(1,1)=ihi
   ipnts(2,1)=jlo
   ipnts(3,1)=klo
c upper point of range
   ipnts(1,2)=ihi
   ipnts(2,2)=jhi
   ipnts(3,2)=khi
   call cg_boco_write_f(index_file,index_base,index_zone,'Ihi',
+    BCExtrapolate,PointRange,2,ipnts,index_bc,ier)
c write boundary conditions for jlo face, defining range first
c (user can give any name)
c lower point of range
   ipnts(1,1)=ilo
   ipnts(2,1)=jlo
   ipnts(3,1)=klo
c upper point of range
   ipnts(1,2)=ihi
   ipnts(2,2)=jlo
   ipnts(3,2)=khi
   call cg_boco_write_f(index_file,index_base,index_zone,'Jlo',
+    BCWallInviscid,PointRange,2,ipnts,index_bc,ier)
      ...  etc...
c close CGNS file
   call cg_close_f(index_file,ier)
```

The zone names (e.g., Ilo) are arbitrary. Note that the variable zonename must be declared as a character variable, and isize and ipnts must be dimensioned appropriately.

The layout of the CGNS file for this example is shown in Fig. 11. Four of the children nodes of ZoneBC_t are left off for clarity.

Reading the boundary conditions can also be easily accomplished using API calls, but we do not show an example of this here. Because there are multiple BC_t children nodes under the ZoneBC_t node, the user must first read in the number of children nodes that exist, then loop through them and retrieve the information from each.

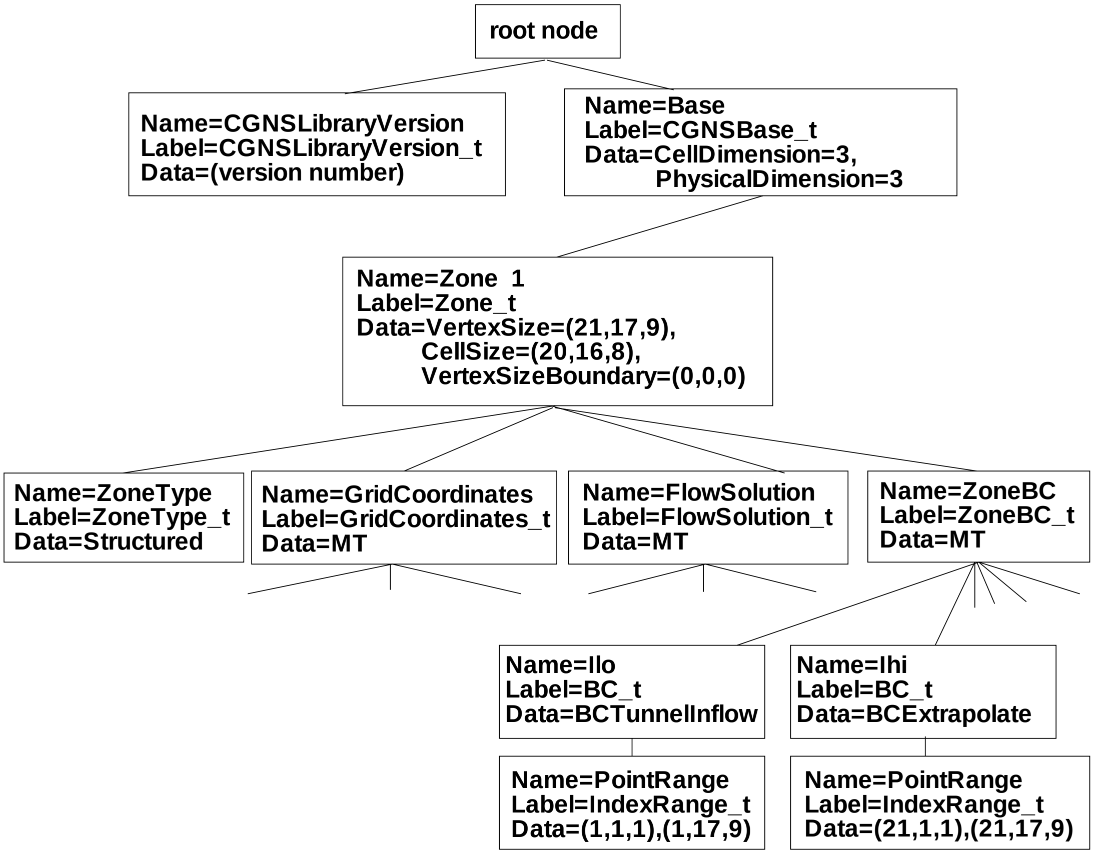

Figure 11: Layout of CGNS file for simple Cartesian structured grid with flow solution and boundary conditions using `PointRange`.

(b) Boundary Conditions Specifying Points

The FORTRAN code segment to write the boundary conditions using PointList is the same as that for PointRange except that the following segment, for example,

```
c write boundary conditions for ilo face, defining range first
c (user can give any name)
   ipnts(1,1)=ilo
   ipnts(2,1)=jlo
   ipnts(3,1)=klo
   ipnts(1,2)=ilo
   ipnts(2,2)=jhi
   ipnts(3,2)=khi
   call cg_boco_write_f(index_file,index_base,index_zone,'Ilo',
+    BCTunnelInflow,PointRange,2,ipnts,index_bc,ier)
```

is replaced by:

```
c write boundary conditions for ilo face, defining pointlist first
c (user can give any name)
   icount=0
   do j=jlo,jhi
      do k=klo,khi
         icount=icount+1
         ipnts(1,icount)=ilo
         ipnts(2,icount)=j
         ipnts(3,icount)=k
      enddo
   enddo
   call cg_boco_write_f(index_file,index_base,index_zone,'Ilo',
+    BCTunnelInflow,PointList,icount,ipnts,index_bc,ier)
```

The layout of the CGNS file in this case is the same as Fig. 11, except that PointRange (IndexRange_t) becomes PointList (IndexArray_t) and there is icount data in the PointList nodes.

2.1.4 Multi-Zone Structured Grid with 1-to-1 Connectivity

For the case of a multi-zone structured grid, each zone is handled individually in the same way as the examples in the preceding sections. However, multi-zone grids also require additional information about how the zones are connected to one another. A discussion of different types of zone-to-zone connectivity can be found in Appendix B. For the example in this section, we show only a simple 1-to-1 connectivity example. We assume that we have a two-zone grid, each identical to the one showed in Fig. 3 ($21 \times 17 \times 9$), except that zone 2 is offset in the x-direction by 20 units. Thus, the *ilo* face of zone 2 abuts the *ihi* face of zone 1, and each abutting point in the two zones touches a point from the neighboring zone. A picture of the grid is shown in Fig. 12.

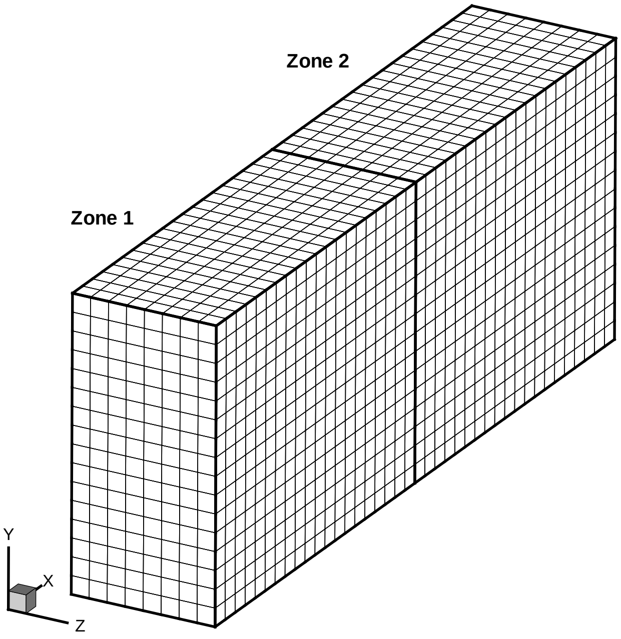

Figure 12: 2-Zone Cartesian structured grid with 1-to-1 connectivity.

The overall layout of this two-zone CGNS file is not shown here. It is similar to those shown earlier, except now there are two zones rather than one. See Appendix B for an additional example.

Now, 1-to-1 connectivity information must be written into each of the zones. There are two ways to record this 1-to-1 information. The first (specific) method is valid only for 1-to-1 interfaces, and the regions *must* be logically rectangular (because they are recorded via `PointRange` and `PointRangeDonor` nodes, for which only two points define the entire region). The second way is more general. It uses `PointList` nodes in combination with `PointListDonor`. (A third method, used to describe interfaces that are *not* point-matched – such as mismatched or overset zones – employs `CellListDonor` and `InterpolantsDonor`.) Refer to the SIDS document [1] for details on the various methods for describing connectivity.

(a) Connectivity Using Specific 1-to-1 Method

The 1-to-1 connectivity information for the current example can be written to a CGNS file using the following FORTRAN code segment (assuming that all grid information has already been written):

```
c WRITE 1-TO-1 CONNECTIVITY INFORMATION TO EXISTING CGNS FILE
   include 'cgnslib_f.h'
c open CGNS file for modify
   call cg_open_f('grid.cgns',CG_MODE_MODIFY,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
c get number of zones (should be 2 for our case)
   call cg_nzones_f(index_file,index_base,nzone,ier)
c loop over zones to get zone sizes and names
   do index_zone=1,nzone
      call cg_zone_read_f(index_file,index_base,index_zone,
   +    zonename(index_zone),isize,ier)
      ilo(index_zone)=1
      ihi(index_zone)=isize(1,1)
      jlo(index_zone)=1
      jhi(index_zone)=isize(2,1)
      klo(index_zone)=1
      khi(index_zone)=isize(3,1)
   enddo
c loop over zones again
   do index_zone=1,nzone
c set up index ranges
      if (index_zone .eq.  1) then
         donorname=zonename(2)
c lower point of receiver range
         ipnts(1,1)=ihi(1)
         ipnts(2,1)=jlo(1)
         ipnts(3,1)=klo(1)
c upper point of receiver range
         ipnts(1,2)=ihi(1)
         ipnts(2,2)=jhi(1)
         ipnts(3,2)=khi(1)
c lower point of donor range
         ipntsdonor(1,1)=ilo(2)
         ipntsdonor(2,1)=jlo(2)
         ipntsdonor(3,1)=klo(2)
c upper point of donor range
         ipntsdonor(1,2)=ilo(2)
         ipntsdonor(2,2)=jhi(2)
         ipntsdonor(3,2)=khi(2)
      else
         donorname=zonename(1)
```

```
c lower point of receiver range
         ipnts(1,1)=ilo(2)
         ipnts(2,1)=jlo(2)
         ipnts(3,1)=klo(2)
c upper point of receiver range
         ipnts(1,2)=ilo(2)
         ipnts(2,2)=jhi(2)
         ipnts(3,2)=khi(2)
c lower point of donor range
         ipntsdonor(1,1)=ihi(1)
         ipntsdonor(2,1)=jlo(1)
         ipntsdonor(3,1)=klo(1)
c upper point of donor range
         ipntsdonor(1,2)=ihi(1)
         ipntsdonor(2,2)=jhi(1)
         ipntsdonor(3,2)=khi(1)
      end if
c set up Transform
      itranfrm(1)=1
      itranfrm(2)=2
      itranfrm(3)=3
c write 1-to-1 info (user can give any name)
      call cg_1to1_write_f(index_file,index_base,index_zone,
   +    'Interface',donorname,ipnts,ipntsdonor,itranfrm,
   +    index_conn,ier)
   enddo
c close CGNS file
   call cg_close_f(index_file,ier)
```

Note that this code segment is geared very specifically toward our 2-zone example, i.e., it relies on our knowledge of this particular case. Transform defines the relative orientation of the i, j, and k indices of the abutting zones. Details concerning the values of Transform are not given here; they can be found in [1]. However, note that Transform values of (1,2,3) indicate that the i, j, k axes of both zones are oriented in the same directions. Reading the connectivity information can also be easily accomplished using API calls, but we do not show an example of this here. And finally, we do not show the layout of the nodes associated with the connectivity here. The interested user is referred to Appendix B for an example figure.

(b) Connectivity Using General Method

Using a more general method, for which each connectivity pair is listed (rather than ranges), the connectivity information for the current example can be written to a CGNS file using the following FORTRAN code segment:

```
c WRITE GENERAL CONNECTIVITY INFORMATION TO EXISTING CGNS FILE
  include 'cgnslib_f.h'
c open CGNS file for modify
  call cg_open_f('grid.cgns',CG_MODE_MODIFY,index_file,ier)
  if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
  index_base=1
c get number of zones (should be 2 for our case)
  call cg_nzones_f(index_file,index_base,nzone,ier)
c loop over zones to get zone sizes and names
  do index_zone=1,nzone
    call cg_zone_read_f(index_file,index_base,index_zone,
  +    zonename(index_zone),isize,ier)
    ilo(index_zone)=1
    ihi(index_zone)=isize(1,1)
    jlo(index_zone)=1
    jhi(index_zone)=isize(2,1)
    klo(index_zone)=1
    khi(index_zone)=isize(3,1)
  enddo
c loop over zones again
  do index_zone=1,nzone
c set up point lists
    if (index_zone .eq.  1) then
      icount=0
      do j=jlo(index_zone),jhi(index_zone)
        do k=klo(index_zone),khi(index_zone)
          icount=icount+1
          ipnts(1,icount)=ihi(1)
          ipnts(2,icount)=j
          ipnts(3,icount)=k
          ipntsdonor(1,icount)=ilo(2)
          ipntsdonor(2,icount)=j
          ipntsdonor(3,icount)=k
        enddo
      enddo
      donorname=zonename(2)
    else
      icount=0
      do j=jlo(index_zone),jhi(index_zone)
        do k=klo(index_zone),khi(index_zone)
          icount=icount+1
          ipnts(1,icount)=ilo(2)
          ipnts(2,icount)=j
          ipnts(3,icount)=k
          ipntsdonor(1,icount)=ihi(1)
          ipntsdonor(2,icount)=j
          ipntsdonor(3,icount)=k
        enddo
      enddo
      donorname=zonename(1)
    end if
c write integer connectivity info (user can give any name)
    call cg_conn_write_f(index_file,index_base,index_zone,
  +   'GenInterface',Vertex,Abutting1to1,PointList,icount,ipnts,
  +   donorname,Structured,PointListDonor,Integer,icount,
  +   ipntsdonor,index_conn,ier)
  enddo
c close CGNS file
  call cg_close_f(index_file,ier)
```

We do not describe the method for recording mismatched (patched) or overset connectivity information in this document; the user is referred to [1] for details. However, note that in such cases the use of `CellListDonor` (along with `InterpolantsDonor`) implies the specification of *cell center indices* on the donor side (these would correspond to element numbers in unstructured zones). The `InterpolantsDonor` information consists of real-valued interpolants.

2.2 Unstructured Grid

This section gives several unstructured grid examples. The user should already be familiar with the information covered in section 2.1, which gives structured grid examples. Because much of the organization of the CGNS files is identical for both grid types, many of the ideas covered in the structured grid section are not repeated again here.

2.2.1 Single-Zone Unstructured Grid

This example uses the exact same grid shown earlier in Fig. 3. However, it is now written as an *unstructured* grid, which is made up of a series of 6-sided elements (cubes in this case). A FORTRAN code segment that uses API calls to write this grid to a CGNS file called `grid.cgns` is shown here (note that it *does not matter* how the nodes are ordered in an unstructured zone, but in this example they are ordered sequentially for simplicity of presentation):

```
c WRITE X, Y, Z GRID POINTS TO CGNS FILE
   include 'cgnslib_f.h'
c open CGNS file for write
   call cg_open_f('grid.cgns',CG_MODE_WRITE,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c create base (user can give any name)
   basename='Base'
   icelldim=3
   iphysdim=3
   call cg_base_write_f(index_file,basename,icelldim,iphysdim,index_base,ier)
c define zone name (user can give any name)
   zonename = 'Zone  1'
c We use the same grid as for the structured example with ni=21,
c nj=17, nk=9.  The variables ni, nj, and nk are still used later,
c for convenience when numbering the unstructured grid elements.
   ni=21
   nj=17
   nk=9
c vertex size (21*17*9 = 3213)
   isize(1,1)=3213
c cell size (20*16*8 = 2560)
   isize(1,2)=2560
c boundary vertex size (zero if elements not sorted)
   isize(1,3)=0
c create zone
   call cg_zone_write_f(index_file,index_base,zonename,isize,
+    Unstructured,index_zone,ier)
c write grid coordinates (user must use SIDS-standard names here)
```

```
   call cg_coord_write_f(index_file,index_base,index_zone,RealDouble,
+    'CoordinateX',x,index_coord,ier)
   call cg_coord_write_f(index_file,index_base,index_zone,RealDouble,
+    'CoordinateY',y,index_coord,ier)
   call cg_coord_write_f(index_file,index_base,index_zone,RealDouble,
+    'CoordinateZ',z,index_coord,ier)
c set element connectivity:
c do all the HEXA_8 elements (this part is mandatory):
c maintain SIDS-standard ordering
   ielem_no=0
c index no of first element
   nelem_start=1
   do k=1,nk-1
      do j=1,nj-1
         do i=1,ni-1
            ielem_no=ielem_no+1
c in this example, due to the order in the node numbering, the
c hexahedral elements can be reconstructed using the following
c relationships:
            ifirstnode=i+(j-1)*ni+(k-1)*ni*nj
            ielem(1,ielem_no)=ifirstnode
            ielem(2,ielem_no)=ifirstnode+1
            ielem(3,ielem_no)=ifirstnode+1+ni
            ielem(4,ielem_no)=ifirstnode+ni
            ielem(5,ielem_no)=ifirstnode+ni*nj
            ielem(6,ielem_no)=ifirstnode+ni*nj+1
            ielem(7,ielem_no)=ifirstnode+ni*nj+1+ni
            ielem(8,ielem_no)=ifirstnode+ni*nj+ni
         enddo
      enddo
   enddo
c index no of last element (=2560)
   nelem_end=ielem_no
c unsorted boundary elements
   nbdyelem=0
c write HEXA_8 element connectivity (user can give any name)
   call cg_section_write_f(index_file,index_base,index_zone,
+    'Elem',HEXA_8,nelem_start,nelem_end,nbdyelem,ielem,
+    index_section,ier)
c close CGNS file
   call cg_close_f(index_file,ier)
```

Note that for unstructured zones, the index dimension is always 1 (because only one index value is required to identify a position in the mesh), so the isize array contains the total vertex size, cell size, and boundary vertex size for the zone. In this example, the `ielem` array must be dimensioned exactly as $(8,N)$, where N is greater than or equal to the total number of elements. The node points that lie in the lower left corner of Fig. 3 are shown schematically for two elements in Fig. 13. Here it can be seen, for example, that node numbers 1, 2, 23, 22, 358, 359, 380, and 379 make up element 1.

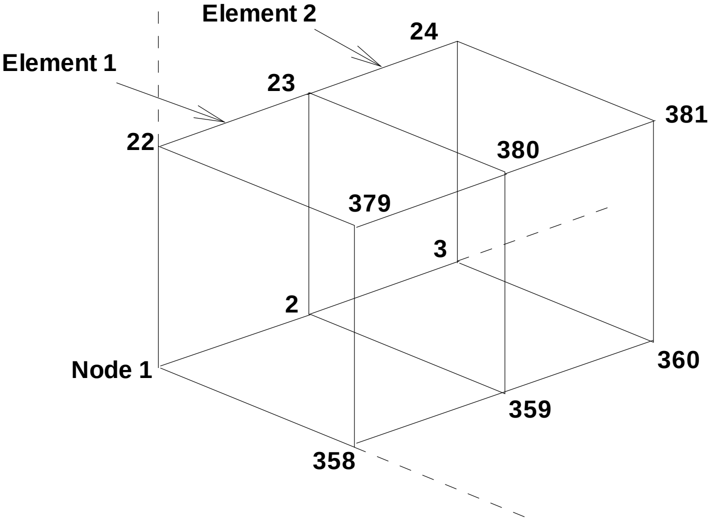

Figure 13: Schematic representation of nodes and elements of unstructured grid.

The overall layout of the CGNS file created by the above code segment is shown in Fig. 14. The nodes for `y` and `z` are left off due to lack of space. Compare this figure with the layout for the structured version of this grid in Fig. 4.

For unstructured zones, the user may also wish to separately list the boundary elements in the CGNS file. This may be useful for assigning boundary conditions, as we will show in section 2.2.3 below. In the current example, assume that the user wishes to assign three different types of boundary conditions: inflow at one end, outflow at the other end, and side walls on the four faces in-between. To accomplish this, it would be helpful to have three additional `Elements_t` nodes in the CGNS file, each of which lists the corresponding faces as elements (`QUAD_4` in this case).

A FORTRAN code segment that accomplishes a part of this is given here. It may be a part of the same code (above) that defined the grid and `HEXA_8` connectivity.

```
c do boundary (QUAD) elements (this part is optional,
c but you must do it if you eventually want to define BCs
c at element faces rather than at nodes):
c INFLOW:
   ielem_no=0
c index no of first element
```

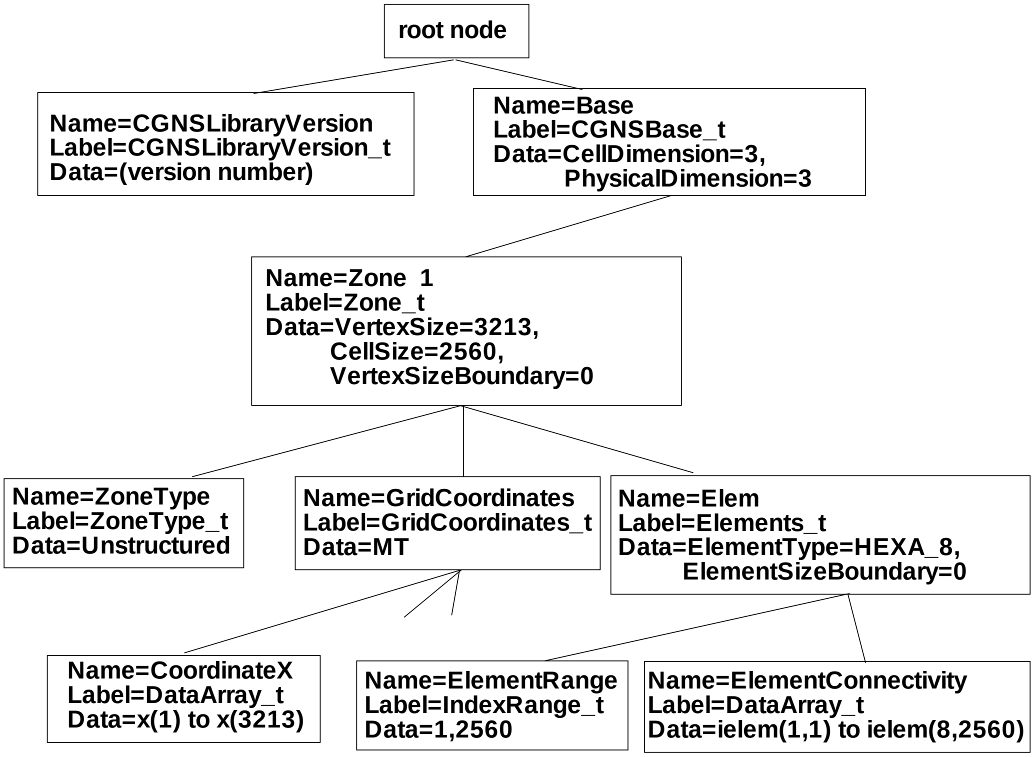

Figure 14: Layout of CGNS file for unstructured grid.

```
   nelem_start=nelem_end+1
   i=1
   do k=1,nk-1
      do j=1,nj-1
         ielem_no=ielem_no+1
         ifirstnode=i+(j-1)*ni+(k-1)*ni*nj
         jelem(1,ielem_no)=ifirstnode
         jelem(2,ielem_no)=ifirstnode+ni*nj
         jelem(3,ielem_no)=ifirstnode+ni*nj+ni
         jelem(4,ielem_no)=ifirstnode+ni
      enddo
   enddo
c index no of last element
   nelem_end=nelem_start+ielem_no-1
c write QUAD element connectivity for inflow face (user can give any name)
   call cg_section_write_f(index_file,index_base,index_zone,
+    'InflowElem',QUAD_4,nelem_start,nelem_end,nbdyelem,
+    jelem,index_section,ier)
c OUTFLOW:
      ...  etc...
```

In this example, the `jelem` array must be dimensioned exactly as (4,N), where N is greater than or equal to the total number of elements. Note that the `nelem_start` and `nelem_end` range is defined *subsequent* to the range of any other elements (i.e., the `HEXA_8` elements) already defined in this zone. In other words, all elements in a given zone must have a different number.

The layout of the CGNS file in this case is exactly the same as that shown in Fig. 14, except that there are now three additional `Elements_t` nodes under `Zone_t`. These are shown separately in Fig. 15.

2.2.2 Single-Zone Unstructured Grid and Flow Solution

To add a flow solution to an unstructured zone, the procedure is identical to that for a structured zone. However, the Rind field for unstructured grids indicates additional points rather than planes. Example of Rind capability for unstructured grids is not covered here. Considering the vertex and cell-center examples shown in section 2.1.2, the only difference for unstructured zones is that all arrays are one-dimensional (there is only one index), as opposed to three indices for 3-D structured arrays. A vertex solution indicates that the solution is stored at vertices or nodes. In the above example, there would be lists of 3213 data array items per solution variable. A cell center solution implies that the solution is stored at the center of each element. In the above example, there would be lists of 2560 data array items per solution variable.

The overall layout of the CGNS file is the same as that shown in Fig. 14, except that

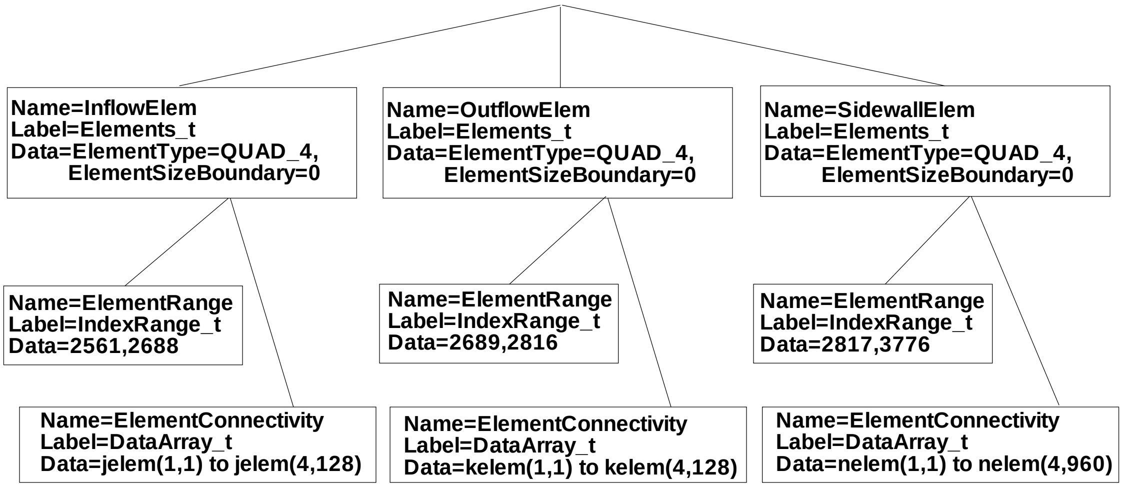

Figure 15: Layout of additional `Elements_t` boundary face nodes.

there would also be a `FlowSolution_t` node under `Zone  1`, and this node would have the children nodes `GridLocation`, `Density`, and `Pressure`.

2.2.3 Single-Zone Unstructured Grid with Boundary Conditions

When writing boundary conditions to a CGNS file for an unstructured zone, one can follow the same general procedure outlined in section 2.1.3 for a structured zone. In other words, the boundary conditions can be defined for point ranges or for individual points, where the points refer to nodes (vertices) of the grid. Coding would be essentially the same as that presented in section 2.1.3, except that the points and/or ranges are now one-dimensional (there is only one index), as opposed to three indices for 3-D structured arrays.

However, for unstructured zones this is generally not the recommended method. Usually, for unstructured zones one has also already defined additional `Elements_t` nodes that *define* the boundary face elements (see section 2.2.1). Therefore, it is best for the boundary conditions to be associated with these elements rather than with the nodes.

Because this concept is quite different from what was done with the structured zone earlier, we illustrate it with an example. At the end of section 2.2.1, we showed how to create the additional `Elements_t` nodes defining the boundary faces. The face-center boundary conditions now can be written using the following code segment.

```
c WRITE BOUNDARY CONDITIONS TO EXISTING CGNS FILE
   include 'cgnslib_f.h'
```

```
c open CGNS file for modify
   call cg_open_f('grid.cgns',CG_MODE_MODIFY,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
c we know there is only one zone (real working code would check!)
   index_zone=1
c we know that for the unstructured zone, the following face elements
c have been defined as inflow (real working code would check!):
   nelem_start=2561
   nelem_end=2688
   icount=0
   do n=nelem_start,nelem_end
      icount=icount+1
      ipnts(icount)=n
   enddo
c write boundary conditions for ilo face
   call cg_boco_write_f(index_file,index_base,index_zone,'Ilo',
+    BCTunnelInflow,PointList,icount,ipnts,index_bc,ier)
c we know that for the unstructured zone, the following face elements
c have been defined as outflow (real working code would check!):
   nelem_start=2689
   nelem_end=2816
   icount=0
   do n=nelem_start,nelem_end
      icount=icount+1
      ipnts(icount)=n
   enddo
c write boundary conditions for ihi face
   call cg_boco_write_f(index_file,index_base,index_zone,'Ihi',
+    BCExtrapolate,PointList,icount,ipnts,index_bc,ier)
c we know that for the unstructured zone, the following face elements
c have been defined as walls (real working code would check!):
   nelem_start=2817
   nelem_end=3776
   icount=0
   do n=nelem_start,nelem_end
      icount=icount+1
      ipnts(icount)=n
   enddo
c write boundary conditions for wall faces
   call cg_boco_write_f(index_file,index_base,index_zone,'Walls',
+    BCWallInviscid,PointList,icount,ipnts,index_bc,ier)
c
c the above are all face-center locations for the BCs - must indicate this,
c otherwise Vertices will be assumed!
   do ibc=1,index_bc
c (the following call positions you in BC_t - it assumes there
c is only one Zone_t and one ZoneBC_t - real working code would check!)
      call cg_goto_f(index_file,index_base,ier,’Zone_t’,1,
+       ’ZoneBC_t’,1,’BC_t’,ibc,’end’)
      call cg_gridlocation_write_f(FaceCenter,ier)
   enddo
c close CGNS file
   call cg_close_f(index_file,ier)
```

Note that we assume here that we know in advance the element numbers associated with each of the boundaries. We have written these element numbers as a PointList, but, because they are in order, we could just as easily have used PointRange instead. In that case, only two ipnts values would be needed, equal to nelem_start and nelem_end, and icount would be 2. Finally, note that the GridLocation_t node under BC_t must be written using the API call cg_goto_f (which positions you correctly in the tree) followed by cg_gridlocation_write_f.

Note that the use of ElementList and ElementRange (recommended in previous versions of CGNS) has been deprecated and should not be used in new code. These are still accepted, but will be internally replaced with the appropriate values of PointList/PointRange and GridLocation_t.

A portion of the layout of the CGNS file for the ZoneBC_t node and its children is shown in Fig. 16. The ZoneBC_t node lies directly under Zone_t. The three figures, Figs. 14, 15, and 16 taken together, constitute the entire layout of the file.

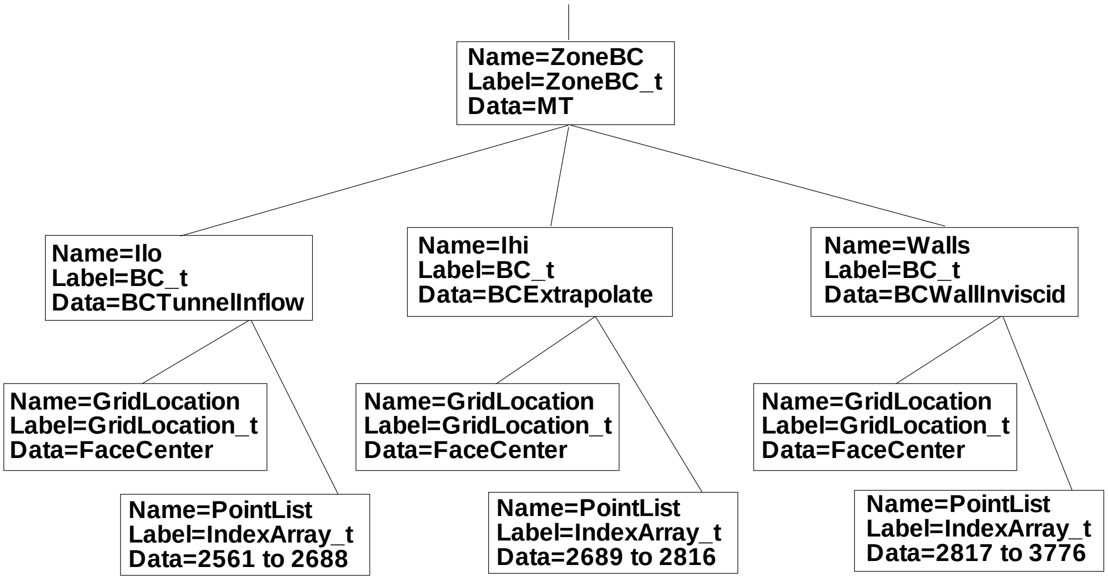

Figure 16: Layout of part of CGNS file for an unstructured zone with boundary conditions defined at face-center elements.

3 ADDITIONAL INFORMATION

This section introduces several additional types of data in CGNS. These items are by no means necessary to include when getting started, but it is likely that most users will eventually want to implement some of them into their CGNS files at some point in the future. The section ends with a discussion on the usage of links.

3.1 Convergence History

The ConvergenceHistory_t node can be used to store data associated with the convergence of a CFD solution. For example, one may wish to store the global coefficient of lift as a function of iterations. In this case, this variable should be stored at the CGNSBase_t level of the CGNS file. This is achieved using the API in the following FORTRAN code segment:

```
c WRITE CONVERGENCE HISTORY INFORMATION TO EXISTING CGNS FILE
   include 'cgnslib_f.h'
c open CGNS file for modify
   call cg_open_f('grid.cgns',CG_MODE_MODIFY,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
```

```
c go to base node
   call cg_goto_f(index_file,index_base,ier,'end')
c create history node (SIDS names it GlobalConvergenceHistory at base level)
c ntt is the number of recorded iterations
   call cg_convergence_write_f(ntt,'',ier)
c go to new history node
   call cg_goto_f(index_file,index_base,ier,'ConvergenceHistory_t',
+    1,'end')
c write lift coefficient array (user must use SIDS-standard name here)
   call cg_array_write_f('CoefLift',RealDouble,1,ntt,cl,ier)
c close CGNS file
   call cg_close_f(index_file,ier)
```

In this example, the array `cl` must be declared as an array of size `ntt` or larger. Additional arrays *of the same size* may also be written under the `ConvergenceHistory_t` node. Note that the call to `cg_convergence_write_f` includes a blank string in this case, because we are not recording norm definitions.

3.2 Descriptor Nodes

Descriptor nodes, which record character strings and can be inserted nearly everywhere in a CGNS file, have many possible uses. Users can insert comments or descriptions to help clarify the content of some data in the CGNS file. In Appendix B, we mention a possible use for descriptor nodes to describe data that is `UserDefinedData_t`. Another potentially desirable use of the descriptor node is to maintain copies of the entire *input* file(s) from the CFD application code. Because descriptor nodes can include carriage returns, entire ASCII files can be "swallowed" into the CGNS file. In this way, a future user can see and retrieve the exact input file(s) used by the CFD code to generate the data contained in the CGNS file. The only ambiguity possible would be whether the CFD code itself has changed since that time; but if the CFD code has strict version control, then complete recoverability should be possible.

An example that writes a descriptor node at the `CGNSBase_t` level is given here:

```
c WRITE DESCRIPTOR NODE AT BASE LEVEL
   include 'cgnslib_f.h'
c open CGNS file for modify
   call cg_open_f('grid.cgns',CG_MODE_MODIFY,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
c go to base node
   call cg_goto_f(index_file,index_base,ier,'end')
```

```
c write descriptor node (user can give any name)
   text1='Supersonic vehicle with landing gear'
   text2='M=4.6, Re=6 million'
   textstring=text1//char(10)//text2
   call cg_descriptor_write_f('Information',textstring,ier)
c close CGNS file
   call cg_close_f(index_file,ier)
```

In this example, the Descriptor_t node is named Information and the character string textstring (which is made up of text1 and text2 with a line feed – char(10) – in-between) is written there. All character strings must be declared appropriately.

3.3 Dimensional Data

The node DataClass_t denotes the class of the data. When data is dimensional, then DataClass_t = Dimensional. The DataClass_t node can appear at many levels in the CGNS hierarchy; precedence rules dictate that a DataClass_t lower in the hierarchy supersedes any higher up.

For dimensional data, one generally is expected to indicate the dimensionality of each particular variable through the use of DataClass_t, DimensionalUnits_t, and DimensionalExponents_t. An example of this is shown in the following code segment in which units are added to the structured grid and cell center flow solution from sections 2.1.1 and 2.1.2.

```
c WRITE DIMENSIONAL INFO FOR GRID AND FLOW SOLN
   include 'cgnslib_f.h'
c open CGNS file for modify
   call cg_open_f('grid.cgns',CG_MODE_MODIFY,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
c we know there is only one zone (real working code would check!)
   index_zone=1
c we know there is only one FlowSolution_t (real working code would check!)
   index_flow=1
c we know there is only one GridCoordinates_t (real working code would check!)
   index_grid=1
c put DataClass and DimensionalUnits under Base
   call cg_goto_f(index_file,index_base,ier,'end')
   call cg_dataclass_write_f(Dimensional,ier)
   call cg_units_write_f(Kilogram,Meter,Second,Kelvin,Degree,ier)
c read fields
```

```
   call cg_nfields_f(index_file,index_base,index_zone,index_flow,
+    nfields,ier)
   do if=1,nfields
      call cg_field_info_f(index_file,index_base,index_zone,
+       index_flow,if,idatatype,fieldname,ier)
      if (fieldname .eq.  'Density') then
         exponents(1)=1.
         exponents(2)=-3.
         exponents(3)=0.
         exponents(4)=0.
         exponents(5)=0.
      else if (fieldname .eq.  'Pressure') then
         exponents(1)=1.
         exponents(2)=-1.
         exponents(3)=-2.
         exponents(4)=0.
         exponents(5)=0.
      else
         write(6,'('' Error!  this fieldname not expected:  '',a32)')
+          fieldname
         stop
      end if
c write DimensionalExponents
      call cg_goto_f(index_file,index_base,ier,'Zone_t',1,
+       'FlowSolution_t',1,'DataArray_t',if,'end')
      call cg_exponents_write_f(RealSingle,exponents,ier)
   enddo
c read grid
   call cg_ncoords_f(index_file,index_base,index_zone,ncoords,ier)
   exponents(1)=0.
   exponents(2)=1.
   exponents(3)=0.
   exponents(4)=0.
   exponents(5)=0.
   do ic=1,ncoords
c write DimensionalExponents
      call cg_goto_f(index_file,index_base,ier,'Zone_t',1,
+       'GridCoordinates_t',1,'DataArray_t',ic,'end')
      call cg_exponents_write_f(RealSingle,exponents,ier)
   enddo
c close CGNS file
   call cg_close_f(index_file,ier)
```

Notice in this example that a DataClass_t node and a DimensionalUnits_t node are placed near the top of the hierarchy, under CGNSBase_t. DataClass_t is specified as Dimensional, and DimensionalUnits_t are specified as (Kilogram, Meter, Second, Kelvin, Degree). These specify that, by and large, the entire database is dimensional with MKS units (anything that is *not* dimensional or *not* MKS units could be superseded at lower levels). Then, for each variable locally, one need only specify the DimensionalExponents, where one exponent is defined for each unit.

The layout of part of the resulting CGNS file from the above example is shown in Fig. 17. The density has units of kilogram/meter^3, and the pressure has units of kilogram/(meter-second^2). The grid coordinates (not shown in the figure) have units of meters.

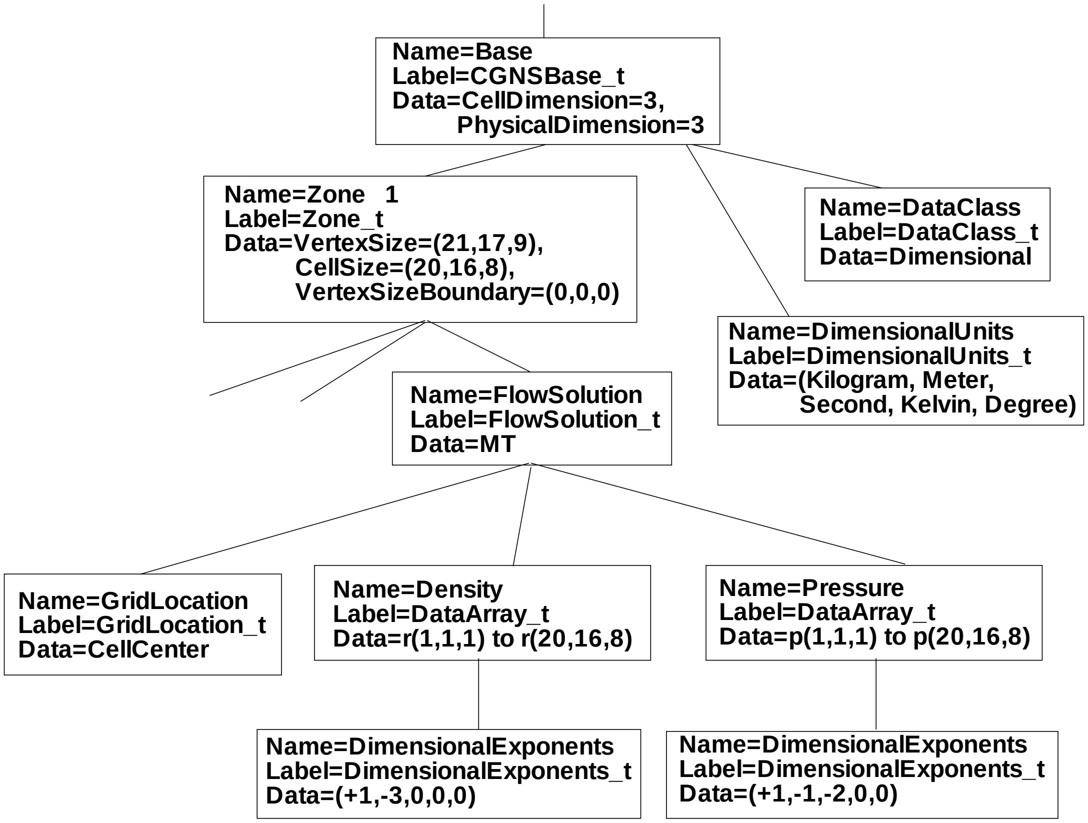

Figure 17: Layout of part of a CGNS file for flow solution at cell centers with dimensional data.

3.4 Nondimensional Data

This example is for the relatively common occurrence of CFD data that is purely nondimensional, for which the reference state is arbitrary (unknown). This type is referred to as NormalizedByUnknownDimensional. Another nondimensional type, NormalizedByDimensional, for which the data is nondimensional but the reference state is *specifically known*, is not covered here.

For a NormalizedByUnknownDimensional database, the DataClass is recorded as such, but also a ReferenceState is *necessary* to define the nondimensionalizations used. (A ReferenceState_t node can be used for any dataset to indicate the global reference state (such as free stream), as well as quantities such as the reference Mach number and Reynolds number. A ReferenceState_t node was not included in section 3.3, but it could have been.)

For the current example, we do not go into detail regarding the choices of the items which should populate the reference state for a NormalizedByUnknownDimensional database. We simply show in the example some typical choices which very often would likely be included. A detailed discussion of how the data in ReferenceState_t defines the nondimensionalizations is given in the SIDS document [1].

```
c WRITE NONDIMENSIONAL INFO
   include 'cgnslib_f.h'
c open CGNS file for modify
   call cg_open_f('grid.cgns',CG_MODE_MODIFY,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
c put DataClass under Base
   call cg_goto_f(index_file,index_base,ier,'end')
   call cg_dataclass_write_f(NormalizedByUnknownDimensional,ier)
c put ReferenceState under Base
   call cg_state_write_f('ReferenceQuantities',ier)
c Go to ReferenceState node, write Mach array and its dataclass
   call cg_goto_f(index_file,index_base,ier,'ReferenceState_t',1,
+    'end')
   call cg_array_write_f('Mach',RealSingle,1,1,xmach,ier)
   call cg_goto_f(index_file,index_base,ier,'ReferenceState_t',1,
+    'DataArray_t',1,'end')
   call cg_dataclass_write_f(NondimensionalParameter,ier)
c Go to ReferenceState node, write Reynolds array and its dataclass
   call cg_goto_f(index_file,index_base,ier,'ReferenceState_t',1,
+    'end')
   call cg_array_write_f('Reynolds',RealSingle,1,1,reue,ier)
   call cg_goto_f(index_file,index_base,ier,'ReferenceState_t',1,
+    'DataArray_t',2,'end')
   call cg_dataclass_write_f(NondimensionalParameter,ier)
c Go to ReferenceState node to write reference quantities:
   call cg_goto_f(index_file,index_base,ier,'ReferenceState_t',1,
+    'end')
c First, write reference quantities that make up Mach and Reynolds:
c Mach_Velocity
   call cg_array_write_f('Mach_Velocity',RealSingle,1,1,xmv,ier)
```

```
c Mach_VelocitySound
   call cg_array_write_f('Mach_VelocitySound',RealSingle,
+    1,1,xmc,ier)
c Reynolds_Velocity
   call cg_array_write_f('Reynolds_Velocity',RealSingle,
+    1,1,rev,ier)
c Reynolds_Length
   call cg_array_write_f('Reynolds_Length',RealSingle,
+    1,1,rel,ier)
c Reynolds_ViscosityKinematic
   call cg_array_write_f('Reynolds_ViscosityKinematic',RealSingle,
+    1,1,renu,ier)
c
c Next, write flow field reference quantities:
c Density
   call cg_array_write_f('Density',RealSingle,1,1,rho0,ier)
c Pressure
   call cg_array_write_f('Pressure',RealSingle,1,1,p0,ier)
c VelocitySound
   call cg_array_write_f('VelocitySound',RealSingle,1,1,c0,ier)
c ViscosityMolecular
   call cg_array_write_f('ViscosityMolecular',RealSingle,
+    1,1,vm0,ier)
c LengthReference
   call cg_array_write_f('LengthReference',RealSingle,
+    1,1,xlength0,ier)
c VelocityX
   call cg_array_write_f('VelocityX',RealSingle,1,1,vx,ier)
c VelocityY
   call cg_array_write_f('VelocityY',RealSingle,1,1,vy,ier)
c VelocityZ
   call cg_array_write_f('VelocityZ',RealSingle,1,1,vz,ier)
c close CGNS file
   call cg_close_f(index_file,ier)
```

In this case, the only information added to the CGNS file is at the `CGNSBase_t` level. Note that `Mach` and `Reynolds` (which are stored under `ReferenceState`) are variables that are known as "`NondimensionalParameter`"s, so they must each contain a `DataClass` child node stating this (the local `DataClass` nodes supersede the overall `NormalizedByUnknownDimensional` data class that holds for everything else).

The layout of the relevant portion of the resulting CGNS file from the above example is shown in Fig. 18. Many of the reference quantities that appear under `ReferenceState_t` have been left out of the figure to conserve space.

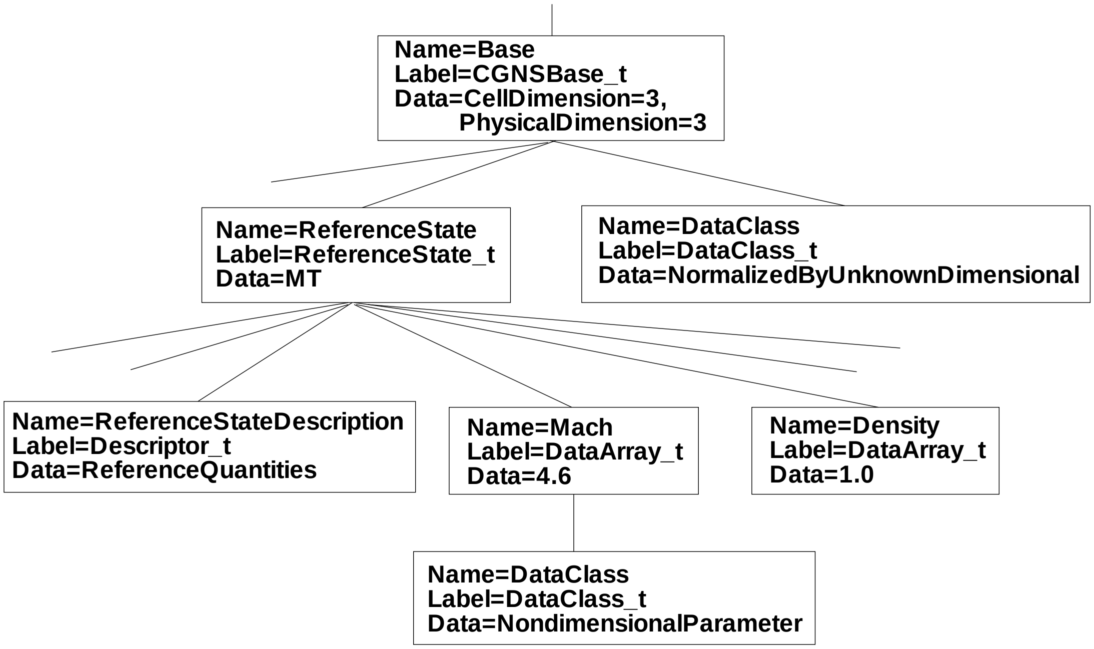

Figure 18: Layout of part of a CGNS file with purely nondimensional data (reference state unknown).

3.5 Flow Equation Sets

The `FlowEquationSet_t` node is useful for describing *how* a flow solution was generated. This is one of the useful self-descriptive aspects of CGNS that may improve the usefulness and longevity of a CFD dataset. For example, under this node, information such as the following may be recorded: the flow field was obtained by solving the thin-layer Navier-Stokes equations (with diffusion only in the j-coordinate direction); the Spalart-Allmaras turbulence model was employed, and an ideal gas assumption was made with $\gamma = 1.4$.

The following FORTRAN code segment writes some of the above example flow equation set information under the `Zone_t` node from our earlier single-zone structured grid example from section 2.1. (Note that a `FlowEquationSet_t` node can also be placed at a higher level, under the `CGNSBase_t` node. The usual precedence rules apply).

```
c WRITE FLOW EQUATION SET INFO
   include 'cgnslib_f.h'
c open CGNS file for modify
   call cg_open_f('grid.cgns',CG_MODE_MODIFY,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
c we know there is only one zone (real working code would check!)
   index_zone=1
```

```
c existing file must be 3D structured (real working code would check!)
c Create 'FlowEquationSet' node under 'Zone_t'
   call cg_goto_f(index_file,index_base,ier,'Zone_t',index_zone,
+    'end')
c equation dimension = 3
   ieq_dim=3
   call cg_equationset_write_f(ieq_dim,ier)
c
c Create 'GoverningEquations' node under 'FlowEquationSet'
   call cg_goto_f(index_file,index_base,ier,'Zone_t',index_zone,
+    'FlowEquationSet_t',1,'end')
   call cg_governing_write_f(NSTurbulent,ier)
c Create 'DiffusionModel' node under 'GoverningEquations'
   call cg_goto_f(index_file,index_base,ier,'Zone_t',index_zone,
+    'FlowEquationSet_t',1,'GoverningEquations_t',1,'end')
   idata(1)=0
   idata(2)=1
   idata(3)=0
   idata(4)=0
   idata(5)=0
   idata(6)=0
   call cg_diffusion_write_f(idata,ier)
c
c Create 'GasModel' under 'FlowEquationSet'
   call cg_goto_f(index_file,index_base,ier,'Zone_t',index_zone,
+    'FlowEquationSet_t',1,'end')
   call cg_model_write_f('GasModel_t',Ideal,ier)
c Create 'SpecificHeatRatio' under GasModel
   call cg_goto_f(index_file,index_base,ier,'Zone_t',index_zone,
+    'FlowEquationSet_t',1,'GasModel_t',1,'end')
   call cg_array_write_f('SpecificHeatRatio',RealSingle,1,1,
+    gamma,ier)
c Create 'DataClass' under 'SpecificHeatRatio'
   call cg_goto_f(index_file,index_base,ier,'Zone_t',index_zone,
+    'FlowEquationSet_t',1,'GasModel_t',1,'DataArray_t',
+    1,'end')
   call cg_dataclass_write_f(NondimensionalParameter,ier)
c close CGNS file
   call cg_close_f(index_file,ier)
```

This particular example is specific to a 3-D structured zone. In an unstructured zone, the use of DiffusionModel is not valid. The layout of the relevant portion of the resulting CGNS file from the above example is shown in Fig. 19.

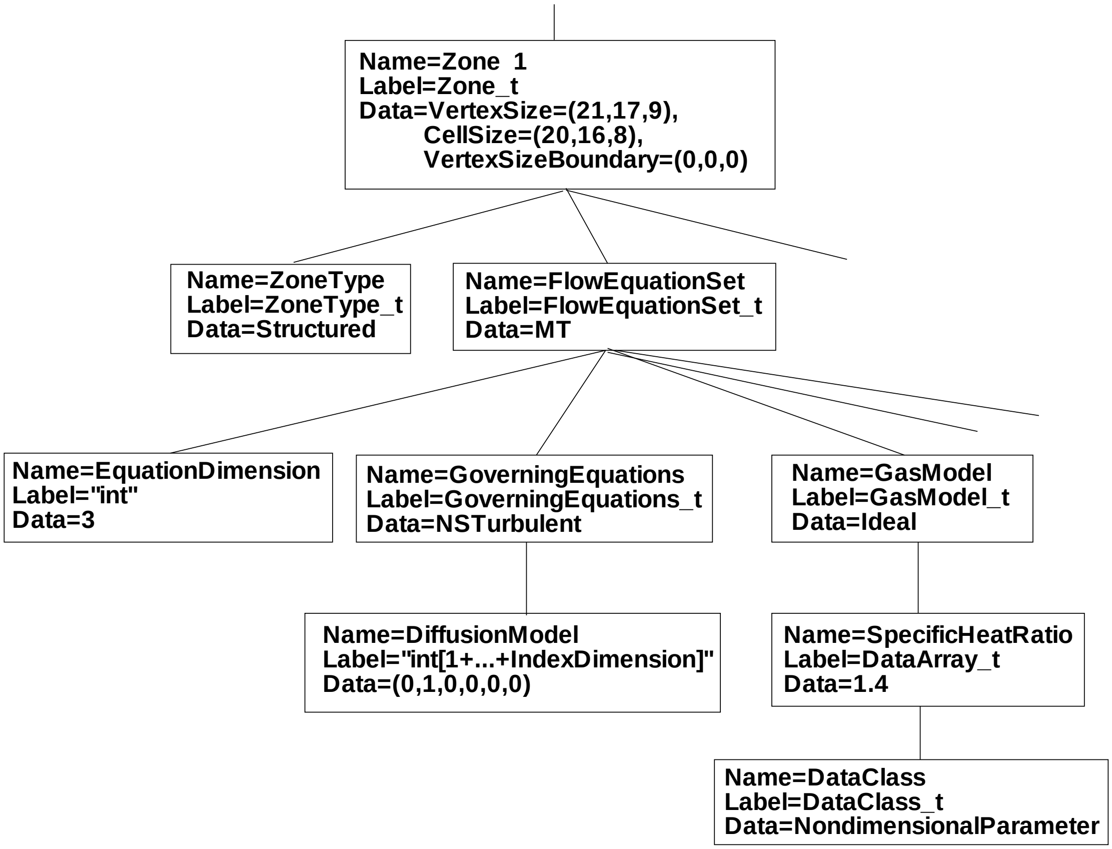

Figure 19: Layout of part of a CGNS file with flow equation set information.

3.6 Time-Dependent Data

Time-dependent data (data with multiple flow solutions) can also be stored in a CGNS file. Different circumstances may produce data with multiple flow solutions; for example:

1. Non-moving grid
2. Rigidly-moving grid
3. Deforming or changing grid

Each of these may either be the result of a time-accurate run, or else may simply be multiple snapshots of a non-time-accurate run as it iterates toward convergence.

This section gives an example for type 1 only. Readers interested in the two other types should refer to the SIDS document [1]. For a non-moving grid, the method for storing the multiple flow solutions is relatively simple: multiple `FlowSolution_t` nodes, each with a different name, are placed under each `Zone_t` node. However, there also needs to be a mechanism for associating each `FlowSolution_t` with a particular time and/or iteration. This is accomplished through the use of `BaseIterativeData_t` (under `CGNSBase_t`) and `ZoneIterativeData_t` (under each `Zone_t`). `BaseIterativeData_t` contains `NumberOfSteps`, the number of times and/or iterations stored, and their values. `ZoneIterativeData_t` contains `FlowSolutionPointers` as a character data array. `FlowSolutionPointers` is dimensioned to be of size `NumberOfSteps`, and contains the *names* of the `FlowSolution_t` nodes within the current zone that correspond with the respective times and/or iterations. Finally, a `SimulationType_t` node is placed under `CGNSBase_t` to designate what type of simulation (e.g., `TimeAccurate`, `NonTimeAccurate`) produced the data. (Note: the `SimulationType_t` node is not restricted for use with time-dependent data; *any* CGNS dataset can employ it!)

The following FORTRAN code segment writes some of the above information, using our earlier single-zone structured grid example from section 2.1. For the purposes of this example, it is assumed that there are 3 flow solutions from a time-accurate simulation, to be output as a function of time to the CGNS file. The variables `r1` and `p1` represent the density and pressure at time 1, `r2` and `p2` are at time 2, and `r3` and `p3` are at time 3.

```
c WRITE FLOW SOLUTION TO EXISTING CGNS FILE
   include 'cgnslib_f.h'
c open CGNS file for modify
   call cg_open_f('grid.cgns',CG_MODE_MODIFY,index_file,ier)
   if (ier .ne.  CG_OK) call cg_error_exit_f
c we know there is only one base (real working code would check!)
   index_base=1
c we know there is only one zone (real working code would check!)
   index_zone=1
c set up the times corresponding to the 3 solutions to be
```

```
c stored:
   time(1)=10.
   time(2)=20.
   time(3)=50.
c define 3 different solution names (user can give any names)
   solname(1) = 'FlowSolution1'
   solname(2) = 'FlowSolution2'
   solname(3) = 'FlowSolution3'
c do loop for the 3 solutions:
   do n=1,3
c create flow solution node
   call cg_sol_write_f(index_file,index_base,index_zone,solname(n),
+    Vertex,index_flow,ier)
c write flow solution (user must use SIDS-standard names here)
   if (n .eq.  1) then
   call cg_field_write_f(index_file,index_base,index_zone,index_flow,
+    RealDouble,'Density',r1,index_field,ier)
   call cg_field_write_f(index_file,index_base,index_zone,index_flow,
+    RealDouble,'Pressure',p1,index_field,ier)
   else if (n .eq.  2) then
   call cg_field_write_f(index_file,index_base,index_zone,index_flow,
+    RealDouble,'Density',r2,index_field,ier)
   call cg_field_write_f(index_file,index_base,index_zone,index_flow,
+    RealDouble,'Pressure',p2,index_field,ier)
   else
   call cg_field_write_f(index_file,index_base,index_zone,index_flow,
+    RealDouble,'Density',r3,index_field,ier)
   call cg_field_write_f(index_file,index_base,index_zone,index_flow,
+    RealDouble,'Pressure',p3,index_field,ier)
   end if
   enddo
c create BaseIterativeData
   nsteps=3
   call cg_biter_write_f(index_file,index_base,'TimeIterValues',
+    nsteps,ier)
c go to BaseIterativeData level and write time values
   call cg_goto_f(index_file,index_base,ier,'BaseIterativeData_t',
+    1,'end')
   call cg_array_write_f('TimeValues',RealDouble,1,3,time,ier)
c create ZoneIterativeData
   call cg_ziter_write_f(index_file,index_base,index_zone,
+    'ZoneIterativeData',ier)
c go to ZoneIterativeData level and give info telling which
c flow solution corresponds with which time (solname(1) corresponds
c with time(1), solname(2) with time(2), and solname(3) with time(3))
```

```
   call cg_goto_f(index_file,index_base,ier,'Zone_t',
+    index_zone,'ZoneIterativeData_t',1,'end')
   idata(1)=32
   idata(2)=3
   call cg_array_write_f('FlowSolutionPointers',Character,2,idata,
+    solname,ier)
c add SimulationType
   call cg_simulation_type_write_f(index_file,index_base,
+    TimeAccurate,ier)
c close CGNS file
   call cg_close_f(index_file,ier)
```

As cautioned for earlier coding snippets, dimensions must be set appropriately for all variables. The variable `time` (which is an array dimensioned size 3 in this case) contains the time values stored under `BaseIterativeData_t`. The layout of the resulting CGNS file from the above example is shown in Fig. 20. Compare this figure with Fig. 6. To conserve space, the `GridCoordinates_t`, `ZoneType_t`, and all nodes underneath `FlowSolution_t` have been left off.

3.7 Using Links

A link associates one node to another within a CGNS tree, or even one node to another in a separate CGNS file altogether. This can be useful when there is repeated data; rather than write the same data multiple times, links can point to the data written only once.

One very important use of links that may be required by many users is to point to grid coordinates. This usage comes about in the following way. Suppose a user is planning to use a particular grid for multiple cases. There are several options for how to store the data. Among these are:

1. Keep a copy of the grid with each flow solution in separate CGNS files.
2. Keep just one CGNS file, with the grid and multiple `FlowSolution_t` nodes; each `FlowSolution_t` node corresponds with a different case.
3. Keep just one CGNS file, with multiple `CGNSBase_t` nodes. The grid and one flow solution would be stored under one base. Other bases would each contain a separate flow solution, plus a link to the grid coordinates in the first base.
4. Keep one CGNS file with the grid coordinates defined, and store the flow solution for each case in its own separate CGNS file, with a link to the grid coordinates.

Item 1 is conceptually the most direct, and is certainly the recommended method in general (this is the way all example CGNS files have been portrayed so far in this document). However, if the grid is very large, then this method causes a lot of storage

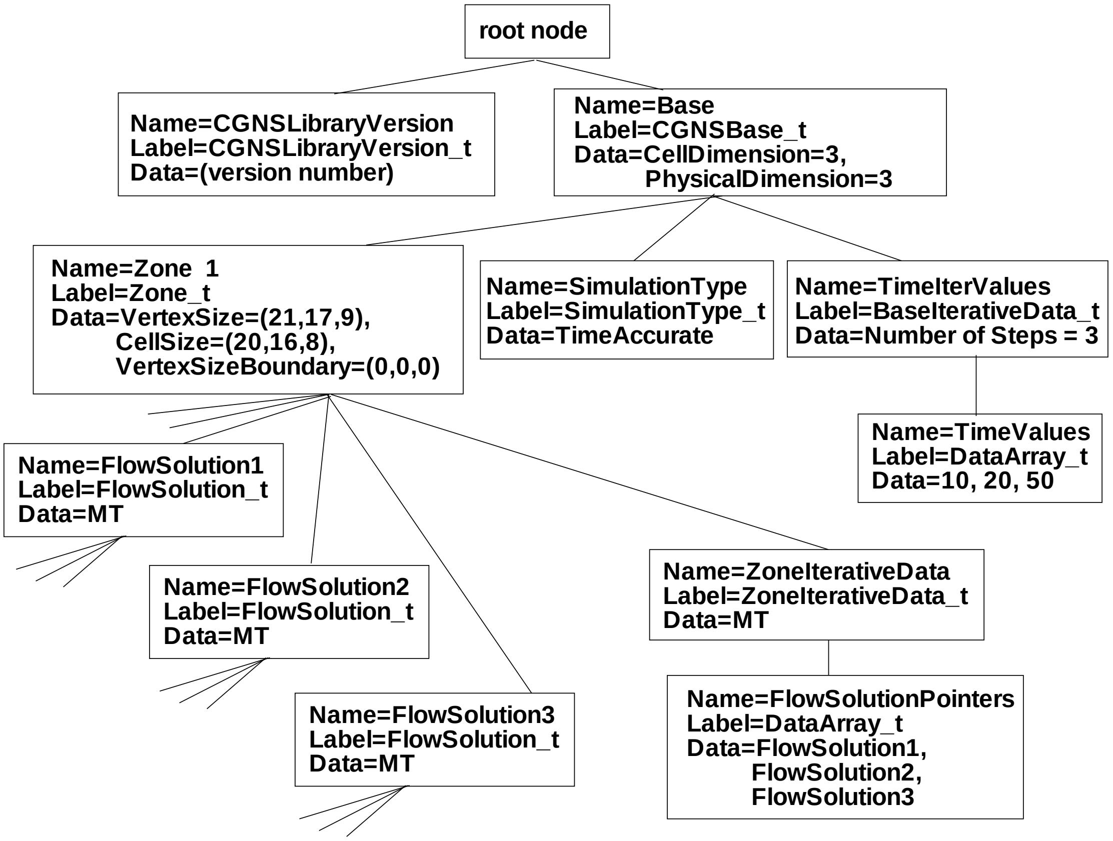

Figure 20: Layout of CGNS file for simple Cartesian structured grid with multiple time-accurate flow solutions (non-moving grid coordinates).

space to be unnecessarily used to store the same grid points multiple times. Item 2 may or may not be a viable option. If the user is striving to have the CGNS file be completely self-descriptive, with `ReferenceState` and `FlowEquationSet` describing the relevant conditions, then this method cannot be used if the `ReferenceState` or `FlowEquationSet` is different between the cases (for example, different Mach numbers, Reynolds numbers, or angles of attack). Item 3 removes this restriction. It uses links to the grid coordinates within the same file. Item 4 is similar to item 3, except that the grid coordinates and each flow solution are stored in separate files altogether.

A sample layout showing the relevant portions of two separate CGNS files for an example of item 4 is shown in Fig. 21. Note that for multiple-zone grids, each zone in FILE 1 in this example would have a separate link to the appropriate zone's grid coordinates in FILE 2.

The CGNS API now has the capability to specify links and to query link information in a CGNS file. Previously this was only possible by use of the ADF core library software. However, when a CGNS file is open for writing and the link creation call is issued, the link information is merely recorded and the actual link creation is deferred until the file is written out upon closing it. Therefore, any attempt to go to a location in a linked file while the CGNS file is open for writing will fail. This problem does not exist when a CGNS file is open for modification as link creation is immediate.

Reading a linked CGNS file presents no difficulties for the API, because links are "transparent." As long as any separate linked files keep their name unchanged, and maintain the same position (within the Unix-directory) relative to the parent file, opening the parent file will automatically access the linked ones.

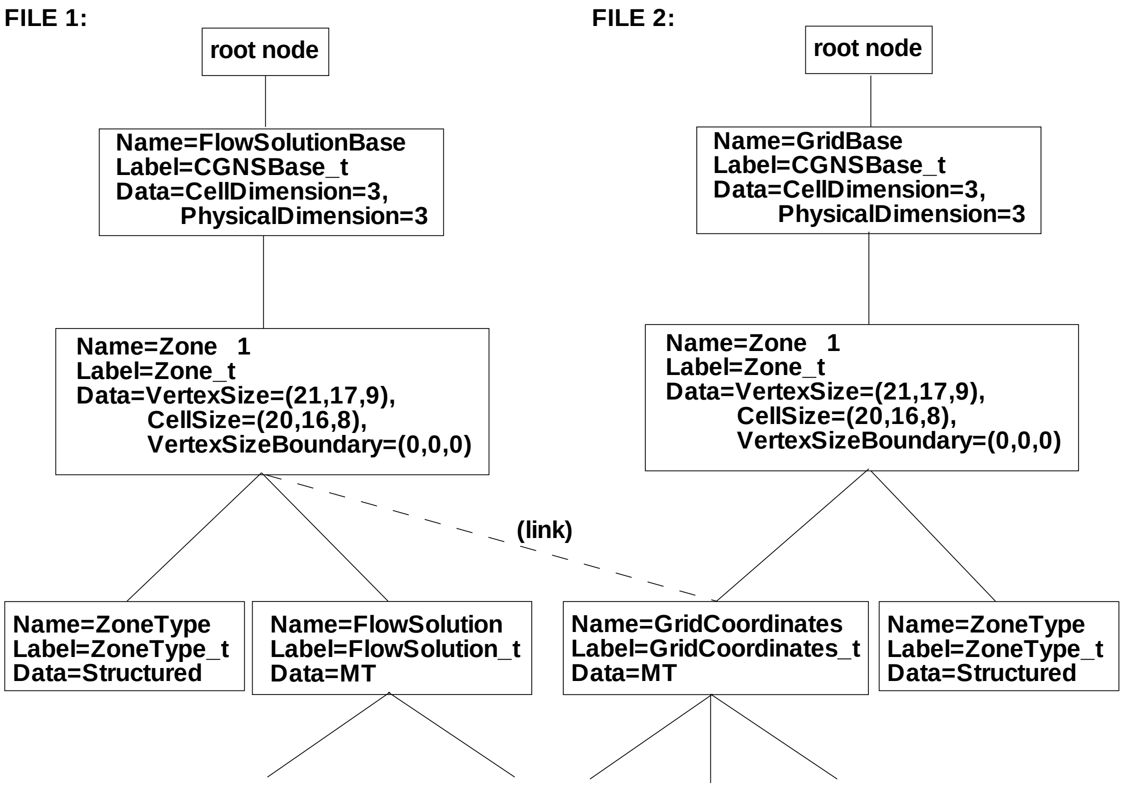

Figure 21: Layout of part of two CGNS files with a link from one to the grid coordinates of the other.

4 TROUBLESHOOTING

4.1 Handling Errors

The API has an extensive number of checks for errors, relating both to illegal usage of ADF as well as relating to SIDS-noncompliance. However, it is not guaranteed that the API will catch all problems prior to reaching the core level. The list of errors that can arise in the ADF core routines themselves are not listed here; they can be found in the file `ADF_interface.c` under "Error strings," and in the ADF User's Guide [2].

If an error occurs, the message given by the ADF or the API routine should hopefully be descriptive enough to point to the source of the error.

4.2 Known Problems

One known problem that can occur, which is not so much a problem as it is a restriction, relates to links. If a user makes a link from one CGNS file to another, then the linked file *must* have write permission if the user wishes to open the linking file in `CG_MODE_MODIFY` or `cg_MODE_WRITE` mode. In other words, opening a CGNS file in `CG_MODE_MODIFY` or `CG_MODE_WRITE` mode implies that the *entire* CGNS hierarchy, including links (since they are transparent), is accessible in that mode.

5 FREQUENTLY ASKED QUESTIONS

Q: Does CGNS support solution array names that are not listed in the SIDS?

A: You can use any data-name that you want for solution arrays. However, if you create a new name not listed in the SIDS, it may not be understood by other applications reading your file.

——————————-

Q: What is a `Family` in CGNS?

A: The families are used to link the mesh to the geometry. The data structure Family_t is optional and can be used to define the geometry of boundary patches by referencing CAD entities. In turn, mesh patches can reference family, so we get: mesh -> family -> geometry.

——————————-

Q: What are `DiscreteData_t` and `IntegralData_t` used for?

A: `DiscreteData_t` can be used to store field data that is not typically considered part of the flow solution `FlowSolution_t`. `IntegralData_t` can be used to store generic global or integral data (a single integer or floating point number is allowed in each `DataArray_t` node under `IntegralData_t`).

——————————-

Q: What are some good programming practices that will help me avoid problems when implementing CGNS in my code?

A: The usual good programming standards apply: use plenty of comments, use logical indentation to make the code more readable, etc. In addition, the API returns an error code from each of its calls; it is a good idea to check this regularly and gracefully exit the program with an error message when it is not zero. In FORTRAN, you can use:

```
if (ier .ne.  0) call cg_error_exit_f
```

If programming in C, you may also assign an error callback function using *cg_error_handler*, which will be called in the case of an error.

——————————-

Q: How can I look at what is in a CGNS file?

A: The utility CGNSview is the best way to look at a CGNS file. This utility allows you to access any node in the file using a Windows-like collapsible node tree. Nodes and data may be added, deleted, and modified. It also has translator capabilities and a crude built-in compliance checker.

——————————-

Q: How can I tell if I have created a truly SIDS-compliant CGNS file?

A: It is currently very difficult to *guarantee* that a user has created a SIDS-compliant

CGNS file, that others can read and understand. But because the API (mid-level-library) has many checks for non-compliance, it is much more difficult for you to make a mistake when using it than if you utilize ADF (core-level) calls. Included with the CGNS distribution is a crude compliance checker, *cgnscheck* located in the tools subdirectory. This may be run from the command line, or from within the CGNSview utility.

Q: How do I write data sets associated with boundary conditions?

A: Writing data sets under boundary conditions is following the "fully SIDS-compliant BC implementation" rather than the "lowest level BC implementation". (See Fig. 24 in Appendix B). In order to do this using the Mid-Level Library, take the following steps:

1. Use `cg_boco_write` to create each `BC_t` node and its associated `BCType` and boundary condition region. (This also creates the top level `ZoneBC_t` node if it does not already exist. Note that only one `ZoneBC_t` node may exist under any given `Zone_t` node.) This is the only step necessary to achieve the "lowest level BC implementation."
2. Use `cg_dataset_write` to create each `BCDataSet_t` node and its associated `BCTypeSimple` under the desired `BC_t` node.
3. Use `cg_bcdata_write` to create each `BCData_t` node (of either type `Dirichlet` or `Neumann`) under the desired `BCDataSet_t` node.
4. Use `cg_goto` to "go to" the appropriate `BCData_t` node.
5. Use `cg_array_write` to write the desired data.

Appendix A. EXAMPLE COMPUTER CODES

The following computer codes are complete, workable versions of the codes mentioned in the text of this User's Guide (plus some that are not mentioned). They can be obtained from the CGNS site at SourceForge (sourceforge.net/projects/cgns). They read and write very simple example CGNS files, in order to help the user understand the CGNS concepts as well as the usage of the API calls. Instructions for compiling them on LINUX systems is contained in comment lines in each program. The following codes are written in both FORTRAN and C.

Note that these programs are very unsophisticated, purposefully for ease of readability. Real working codes would be written more generally, with more checks, and would *not* be as hardwired for particular cases. The codes are listed here by corresponding section.

STRUCTURED GRID

Section 2.1.1:

write_grid_str.f	writes grid
write_grid_str.c	writes grid (C-program example)
read_grid_str.f	reads grid

Section 2.1.2:

write_flowvert_str.f	writes vertex-based flow solution
read_flowvert_str.f	reads vertex-based flow solution
write_flowcent_str.f	writes cell centered flow solution
read_flowcent_str.f	reads cell centered flow solution
write_flowcentrind_str.f	writes cell centered flow solution with rind cells
read_flowcentrind_str.f	reads cell centered flow solution with rind cells

Section 2.1.3:

write_bc_str.f	writes PointRange boundary condition patches
read_bc_str.f	reads PointRange boundary condition patches
write_bcpnts_str.f	writes PointList boundary condition patches
read_bcpnts_str.f	reads PointList boundary condition patches

Section 2.1.4:

write_grid2zn_str.f	writes 2-zone grid
read_grid2zn_str.f	reads 2-zone grid
write_con2zn_str.f	writes 1-to-1 connectivity for 2-zone example
read_con2zn_str.f	reads 1-to-1 connectivity for 2-zone example
write_con2zn_genrl_str.f	writes general 1-to-1 connectivity for 2-zone example
read_con2zn_genrl_str.f	reads general 1-to-1 connectivity for 2-zone example

UNSTRUCTURED GRID

Section 2.2.1:

write_grid_unst.f	writes grid
read_grid_unst.f	reads grid

Section 2.2.2:

write_flowvert_unst.f	writes vertex-based flow solution
read_flowvert_unst.f	reads vertex-based flow solution

Section 2.2.3:

write_bcpnts_unst.f	writes PointList boundary condition patches (FaceCenter)
read_bcpnts_unst.f	reads PointList boundary condition patches (FaceCenter)

GENERAL

Section 3.1:

write_convergence.f	writes convergence history
read_convergence.f	reads convergence history

Section 3.2:

write_descriptor.f	writes descriptor node under CGNSBase_t
read_descriptor.f	reads descriptor node under CGNSBase_t

Section 3.3:

write_dimensional.f	writes dimensional data to an existing grid + flow solution
read_dimensional.f	reads dimensional data from an existing grid + flow solution

Section 3.4:

write_nondimensional.f	writes nondimensional data to an existing CGNS file
read_nondimensional.f	reads nondimensional data from an existing CGNS file

Section 3.5:

write_floweqn_str.f	writes flow equation information for structured example
read_floweqn_str.f	reads flow equation information for structured example

Section 3.6:

write_timevert_str.f	writes time-dependent flow soln (as Vertex) for structured example
read_timevert_str.f	reads time-dependent flow soln (as Vertex) for structured example

Appendix B. OVERVIEW OF THE SIDS

B.1 The Big Picture

As mentioned in the Introduction, a CGNS file 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. Each node is identified by both a label and a name. Most node *labels* are given by a series of characters followed by "_t". There are generally very strict rules governing the labeling conventions in a CGNS file. Node *names* are sometimes user-defined, but sometimes must also follow strict naming conventions. The label identifies a "*type*." For example, `Zone_t` identifies a Zone-type node, and `DataArray_t` identifies a type of node that contains a data array. The name identifies a specific instance of the particular node type. For example, `Density` is the name of a node of type `DataArray_t` that contains an array of densities.

As you become more familiar with how CGNS files are organized, you will notice that, generally, the higher you are in the CGNS hierarchy, the more important the label is (names tend to be user-defined); whereas the lower you are in the hierarchy, the more important the name is. This convention arises because at the higher levels, the broader categories are established, and are used to determine "where to go" in the hierarchy. At the lower levels, the category becomes less important because this is the region where you are searching for specific items.

Throughout the remainder of this first section, we will primarily be referring to the nodes by their *label*, because we are focusing on the "big picture." In later sections, as we get into specific examples, both names and labels will be referred to.

It is important to note at this point that the SIDS document specifies the layout of the CGNS file, in terms of parents and children. However, when a given piece of information is listed as being "under" a node, there are actually two possibilities: the information can be stored *as data in the current node*, or it can be stored *as data in or under a separate child node*. This distinction is illustrated in Fig. 22. The SIDS-to-ADF mapping document [3] determines which of the two possibilities are used for each situation, and must always be consulted along with the SIDS document. (Note there is also a SIDS-to-HDF5 document available.) Throughout the remainder of this appendix, the location of information (whether as data or as a separate child node) will always be explicitly specified, according to the SIDS-to-ADF mapping document.

The remainder of this appendix attempts to summarize the most important and most commonly-used aspects of the SIDS. It does not cover all possible nodes or situations. It is intended as a general overview only. It is also likely that future extensions to the SIDS will add additional capabilities beyond what we cover here.

The top, or entry-level, of the CGNS file is always what is referred to as the "root node." Children to be found directly under this node are the node `CGNSLibraryVersion_t` and one or more `CGNSBase_t` nodes. The `CGNSLibraryVersion_t` node has, as its data,

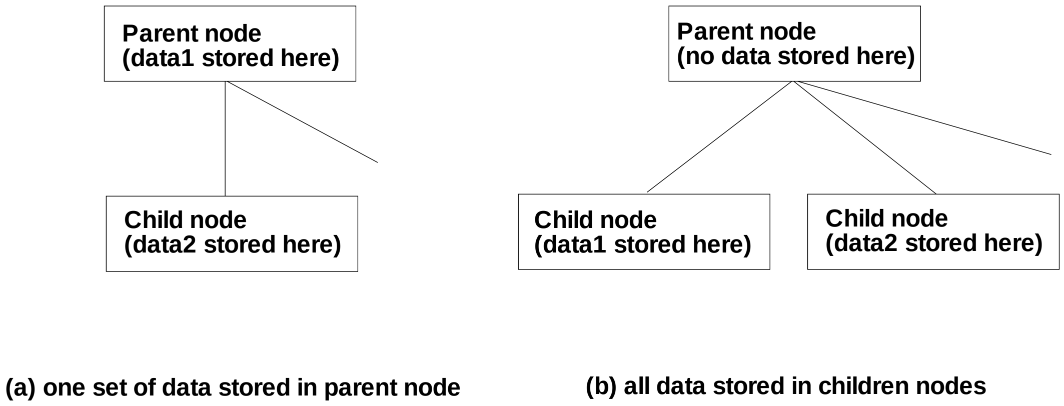

Figure 22: Two possible treatments of a parent node with sets of data "under" it.

the version (release) number of the CGNS standard as defined by the SIDS. The CGNSBase_t node represents the top level for a given database, or "case." Most CGNS files will only have one CGNSBase_t node, although the SIDS allows for any number in order to remain extensible and to allow for the possibility of having more than one "case" in a single file. Here, the definition of "case" is left open. For the remainder of this appendix, we assume that there is only one CGNSBase_t node within a given CGNS file.

The CGNSBase_t node may have, as its children, the following nodes: Zone_t, ConvergenceHistory_t, BaseIterativeData_t, SimulationType_t, Family_t, IntegralData_t, DataClass_t, FlowEquationSet_t, DimensionalUnits_t, ReferenceState_t, Axisymmetry_t, RotatingCoordinates_t, Gravity_t, UserDefinedData_t, and Descriptor_t.

The Zone_t node gives information about a particular zone of the grid; most of the data in the CGNS file is usually found under this node. Any number of Zone_t nodes is allowed at this level. Its children will be described in greater detail below. ConvergenceHistory_t contains solution history information typically output by many CFD codes, such as residual, lift, drag, etc. as a function of iteration number. By convention, its name is GlobalConvergenceHistory. A ConvergenceHistory_t node can exist under the Zone_t node as well, but there, its name is by convention ZoneConvergenceHistory. BaseIterativeData_t stores information relating to the times and/or iteration numbers for a database in which flow solutions and/or grids at multiple times are stored. SimulationType_t describes the type of simulation stored (i.e., TimeAccurate or NonTimeAccurate). Family_t is generally used to tie the grid to geometric CAD data, or to link certain entities together as a common part (e.g., "wing," "strut," etc.). Any number of Family_t nodes is allowed. Axisymmetry_t, RotatingCoordinates_t, and Gravity_t are used for specific situations; details can be found in the SIDS.

The remaining nodes allowed under CGNSBase_t are somewhat more generic, and can exist at other levels in the hierarchy beside this one. They are briefly described here. IntegralData_t is a "catch-all" node for storing any desired sets of generic data. Any number of IntegralData_t nodes is allowed at this level. DataClass_t (which, by convention, has the name DataClass) indicates the form that the data in the CGNSBase_t is stored, for example: Dimensional, NormalizedByDimensional, or NormalizedByUnknownDimensional. FlowEquationSet_t (which, by convention, has the name FlowEquationSet) defines the equations used in the CFD simulation. DimensionalUnits_t (which, by convention, has the name DimensionalUnits) defines the dimensional units used (if any). ReferenceState_t (which, by convention, has the name ReferenceState) defines a reference state. This node is where quantities such as Reynolds number, Mach number, and other reference quantities that define the flow field conditions and/or the nondimensionalizations are stored. UserDefinedData_t is used to store user-defined data that is (by definition) not part of the SIDS standard. Finally, Descriptor_t is used to store descriptor strings. Any number of Descriptor_t nodes is allowed at this level.

The data stored within the CGNSBase_t node itself are the CellDimension and the PhysicalDimension. The CellDimension is the dimensionality of the cells in the mesh (e.g., 3 for volume cell, 2 for face cell). The PhysicalDimension is the number of coordinates required to define a node position (e.g., 1 for 1-D, 2 for 2-D, 3 for 3-D). The index dimension, which is the number of different indices required to reference a node (e.g., 1=i, 2=i,j, 3=i,j,k), is not stored, but can be determined for each zone based on its type (Structured or Unstructured). If Structured, the index dimension is the same as CellDimension. If Unstructured, the index dimension is 1.

Much information can be stored under Zone_t. Because this is an overview, we do not go through it all here. Instead, we only highlight the features that most users are likely to use. ZoneType_t (which, by convention, has the name ZoneType) stores the name Structured or Unstructured. GridCoordinates_t is the parent node of the grid coordinates arrays, such as CoordinateX, CoordinateY, and CoordinateZ. Any number of GridCoordinates_t nodes are allowed at this level (to handle the case of deforming grids). By convention, the original grid coordinates has the name GridCoordinates. FlowSolution_t stores under it nodes which contain the flow solution; for example, Density, VelocityX, VelocityY, VelocityZ, and Pressure. It also gives the location at which the solution is stored (e.g., CellCenter, Vertex), and includes the possibility for including Rind (ghost cell or ghost point) information. Any number of FlowSolution_t nodes are allowed at this level. The Elements_t data structure holds unstructured element data such as connectivity, neighbors, etc. Any number of Elements_t nodes are allowed at this level. ZoneIterativeData_t stores information necessary for a database in which flow solutions at multiple times are stored. Other important nodes under Zone_t are ZoneBC_t (which, by convention, has the name ZoneBC) and ZoneGridConnectivity_t (which, by convention, has the name ZoneGridConnectivity). These store the boundary conditions and the grid connectivity information, respectively. More will be said about these nodes later.

The data stored within the `Zone_t` node itself are the `VertexSize`, the `CellSize`, and the `VertexSizeBoundary`. These are dimensioned by the index dimension, and give the number of vertices, the number of cells, and the number of boundary vertices (used for sorted elements in unstructured zones only), respectively.

An important point to note here is that the API sorts the `Zone_t` nodes alphanumerically according to their *name* when it reads them. This was deemed necessary because most CFD codes currently perform operations on the zones of multiple-zone grids in a certain order. To duplicate existing non-CGNS applications, it is necessary to insure that zones can be read in the desired sequence. (ADF does not necessarily retrieve data in the same order in which it was stored, so the API reader for zones was built to do this.) Hence, when *naming* zones, the user should make sure they are named alphanumerically (if an ordering is desired).

For example, the naming convention `ZoneN`, where N is the zone number, is alphanumeric only up to `Zone9`. `Zone10` through `Zone19` would get sorted between `Zone1` and `Zone2`, and so on. Spaces are allowed in names, so `Zone  N`, with two spaces, (e.g., `Zone  1`, `Zone  2`,... `Zone 99`, `Zone100`,...) is alphanumeric up to `Zone999`. Other zone naming conventions are certainly possible, and are completely up to the user to define appropriately.

A summary graphic of the overall layout of a typical CGNS file is given in Fig. 23. This figure shows the hierarchical data structure, and the relative locations of the nodes. It also indicates (informally) what data, if any, is stored *within* each node. Note that all possible nodes are *not* included here. In particular, note that `Elements_t` nodes are not shown under `Zone_t`; the `Elements_t` nodes would be present for an unstructured zone. Also note that nodes that occur under `ZoneBC_t` and `ZoneGridConnectivity_t` have been omitted; these will also be covered below. Optional nodes such as `SimulationType_t` (under `CGNSBase_t`) are not included. And finally, note that multiple `GridCoordinates_t` and `FlowSolution_t` nodes are allowed, but we show in the figure only one of each. Multiple `FlowSolution_t` nodes are usually only used in the situation when multiple times of time-accurate data are stored, and multiple `GridCoordinates_t` nodes are used for deforming grids.

B.2 Implementation at the Lower Levels of the Hierarchy

Most of the actual data is at the lower levels of the CGNS hierarchy. We do not go into great detail here; the examples in the main body of this document serve as instruction for this. However, there are several general items of importance related to the storage of data that are appropriate to mention here.

Many specific items, variables, and conditions that relate to CFD data are specified in the SIDS. These are standardized *names* that must be used in order that other users will understand what is in your CGNS file. For example, the static density must be called `Density`. Any other name may not be recognized by other users. In fact, if another

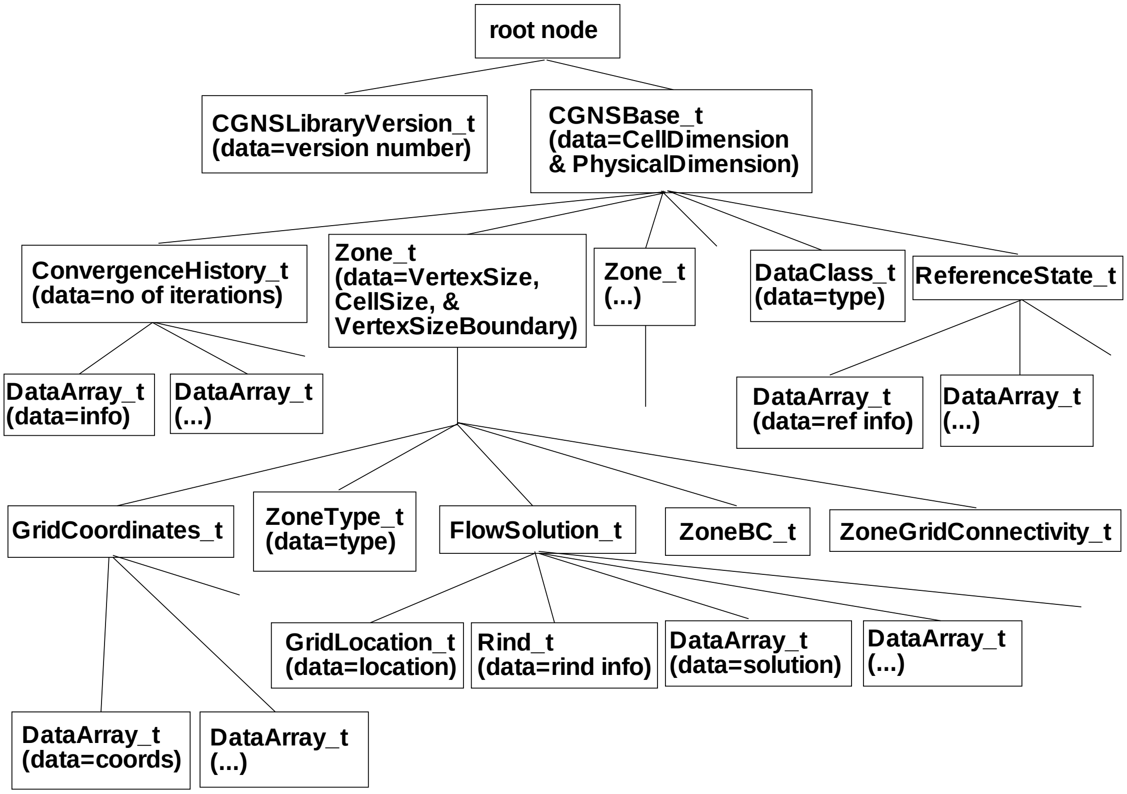

Figure 23: Hierarchical structure of a typical CGNS file (structured grid type).

application code expects "Density," but you name it "density" (lower case "d"), then chances are the other code's search will fail.

Naturally, the items listed in the SIDS cannot cover all possible items required by users. Hence, the SIDS allows for the use of the type UserDefinedData_t for any special type not covered. For example, there are currently only a limited number of defined names for turbulence models in the SIDS (e.g., OneEquation_SpalartAllmaras). As everyone knows, there are a *huge* number of turbulence models and turbulence model variants that exist, so that the SIDS cannot hope to define standardized names for all of them. The type UserDefinedData_t covers this situation.

When UserDefinedData_t is used, however, the user runs the risk that others will be unable to interpret the CGNS file. We therefore recommend that whenever a UserDefinedData_t type is unavoidable, the user also include a companion Descriptor_t node to specify what was done.

It is possible that, if certain items are found to be used more heavily as time goes on, that standardized names may be created and added to the SIDS in the future.

B.3 Boundary Conditions

The boundary conditions hierarchical structure in CGNS can appear to be somewhat daunting at first. Because the CGNS team decided to make the boundary condition information as descriptive as possible and easily extensible to complex situations, there are many layers possible in the hierarchy, and the usage rules can become complex.

However, the SIDS allows for use of simplified versions of the ZoneBC_t node, which are easier to understand and adopt. Essentially, the simplified versions "cut off" the hierarchy at a higher level than the full-blown SIDS boundary condition description. The implication of this is that application codes that use a simplified version must interpret what is meant by each particular boundary condition type, without the help of the CGNS file.

For example, the boundary condition type BCFarfield indicates a boundary condition applied to a far field boundary. Most CFD codes have this type, which performs different functions depending upon whether the local flow field is inflow or outflow, subsonic or supersonic. The full-blown SIDS description of BCFarfield attempts to describe in some detail the methodology involved in this boundary condition. However, if the user chooses to use the minimal "cut off" version, the only information regarding the function of the boundary condition that is stored in the CGNS file is the *name* BCFarfield. An application code must determine from this name alone what is meant.

Example hierarchical structures for both the simplest implementation as well as the full-blown implementation of the ZoneBC_t node are shown in Fig. 24. (These hierarchies make use of an IndexRange_t node. It is also possible to use an IndexArray_t, which gives a complete list of boundary indices or elements, rather than a range.) Note that

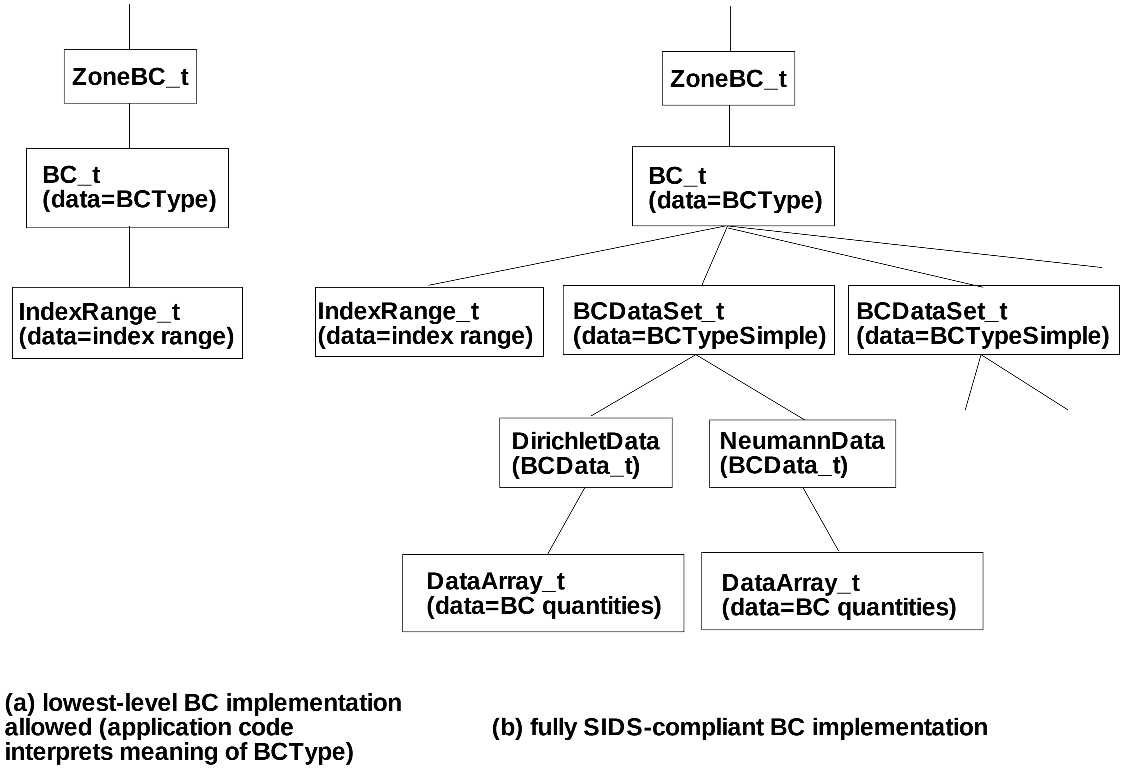

Figure 24: General hierarchical structure of ZoneBC_t.

an intermediate structure, where BCDataSet_t and BCTypeSimple_t are both given but DirichletData and NeumannData are not, is also allowed.

Many boundary condition types are currently defined in the SIDS, but they by no means cover all possible boundary conditions. The type UserDefinedData_t can be used for any special type not covered that the user finds impossible to describe using the existing SIDS. When UserDefinedData_t is used, a companion descriptor node is helpful to describe what was done.

B.4 Zone Connectivity

It is often desirable to specify zone connectivity information when parts of a zone connect with parts of another zone or itself. The connectivity information tells how zones fit together or how a zone twists to reconnect with itself; the information is needed by most CFD flow solvers.

There are three types of connectivity that can occur: point-by-point, patched, and overset. The point-by-point, or 1-to-1, type occurs when the edges of zones abut, and grid vertices from one patch exactly correspond with grid vertices from the other, with no points missing a partner. The patched type occurs when the edges of zones abut, but there is not a correspondence of the points, or they are not partnered with another point. The overset type occurs when zones overlap one another (or a zone overlaps itself).

The SIDS allows for the specification of each of these types of zone connectivity under the ZoneGridConnectivity_t node. All three types can be implemented through the general GridConnectivity_t subnode (overset also requires the use of OversetHoles_t nodes). However, the 1-to-1 type can also utilize, in certain circumstances, the more specific GridConnectivity1to1_t subnode.

Fig. 25 shows a sample hierarchy starting at the ZoneGridConnectivity_t node, for a 1-to-1 type of interface using a GridConnectivity1to1_t subnode. Note in this figure that we now list the name, label, and data within each node. For this structure, the naming convention at the bottom level is particularly important, and is actually more descriptive than the labels. In fact, the label for the Transform node is very strange, and does not even follow the usual "_t" convention. As can be seen in the figure, multiple nodes are allowed under the ZoneGridConnectivity_t node. These can be any combination of GridConnectivity1to1_t, GridConnectivity_t, OversetHoles_t, or Descriptor_t nodes.

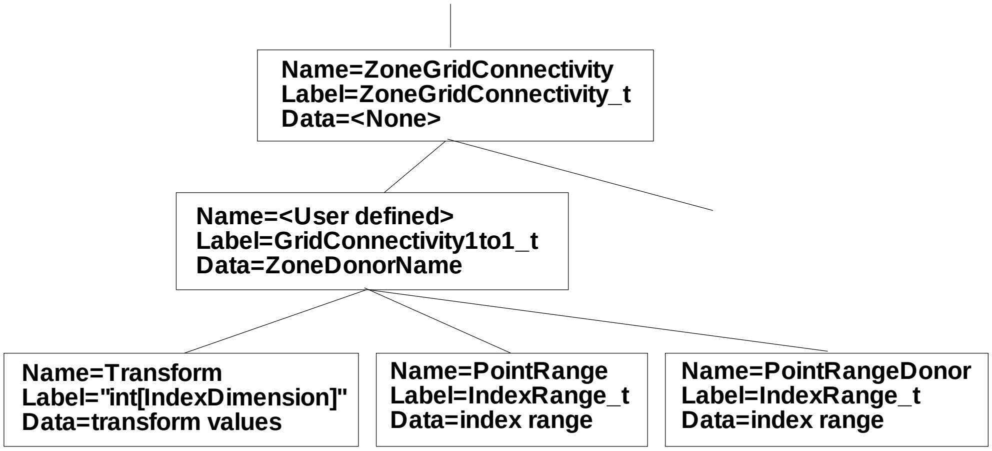

Figure 25: Hierarchical structure of ZoneGridConnectivity_t for a 1-to-1 interface.

A sample hierarchy (again starting at the ZoneGridConnectivity_t node) is shown in Fig. 26 for an *overset* interface using a GridConnectivity_t subnode. The case for a *patched* interface would look the same, except there would be no OversetHoles_t node or its children and GridConnectivityType would be Abutting. Note that CellListDonor and InterpolantsDonor are used for patched or overset interfaces. (PointListDonor can be used in their place if the interface is 1-to-1.) See [1] [3] for details.)

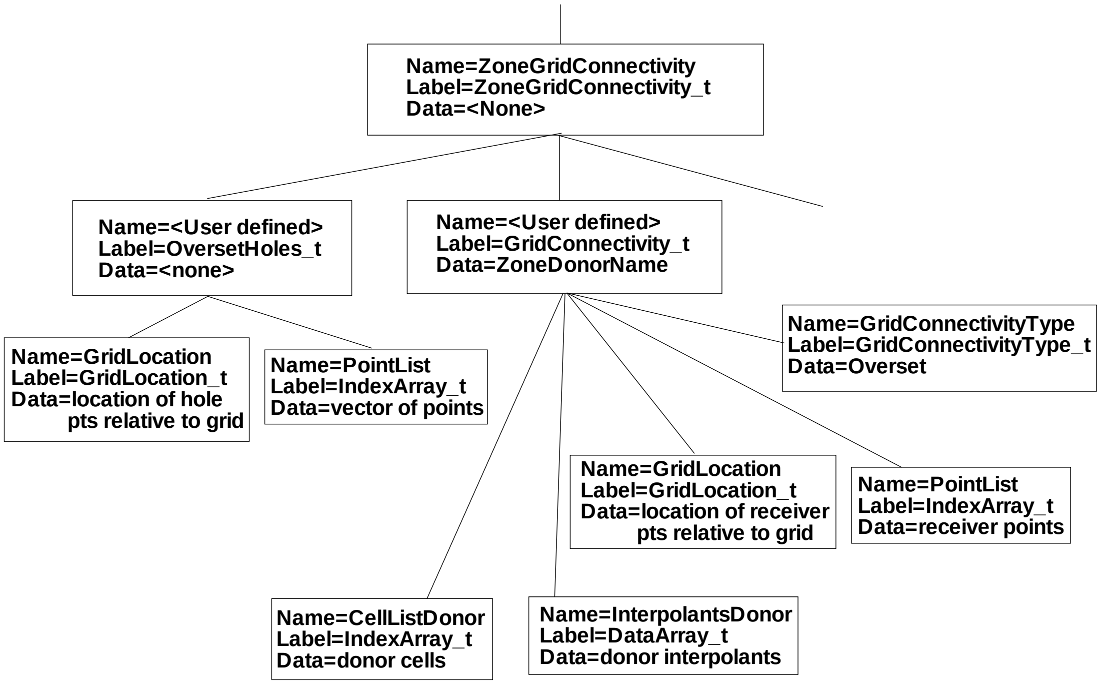

Figure 26: Hierarchical structure of ZoneGridConnectivity_t for an overset interface.

B.5 Structured Zone Example

The following is an example for a structured grid. It corresponds with example 8-A in the SIDS document [1]. It is a 3-D two-zone case, where the two zones are connected in a 1-to-1 fashion at one of each of their faces. Zone 1 is $9 \times 17 \times 11$ and zone 2 is $9 \times 17 \times 21$. The k-max face of zone 1 abuts the k-min face of zone 2.

The hierarchy is shown in Figs. 27 through 30. Only directly relevant parts of the hierarchy are shown here for clarity. For example, DataClass_t, ReferenceState_t, ConvergenceHistory_t, FlowEquationSet_t, and ZoneBC_t have all been left off. However, these (and other) items are *not* required, and the figure still represents a valid SIDS-compliant CGNS file. Note that a data type of MT indicates that there is no data stored in the node.

In this example, the flow solution in zone 1 is given at cell centers, whereas the flow solution in zone 2 is given at the vertices (see Fig. 29). In other words, the zone 1 solution points *do not* correspond with the grid points (as they do in zone 2). They are defined *within* the volumes surrounded by the grid points. This example is constructed this way for the purpose of illustration, but it is unusual; typically one would use only a single flow solution data location for the entire file.

This example also illustrates the use of the Rind_t node, and how it affects the data arrays under a FlowSolution_t. A rind node under FlowSolution_t is used to indicate that the flow solution is outputting additional rind or "ghost" data outside one or more

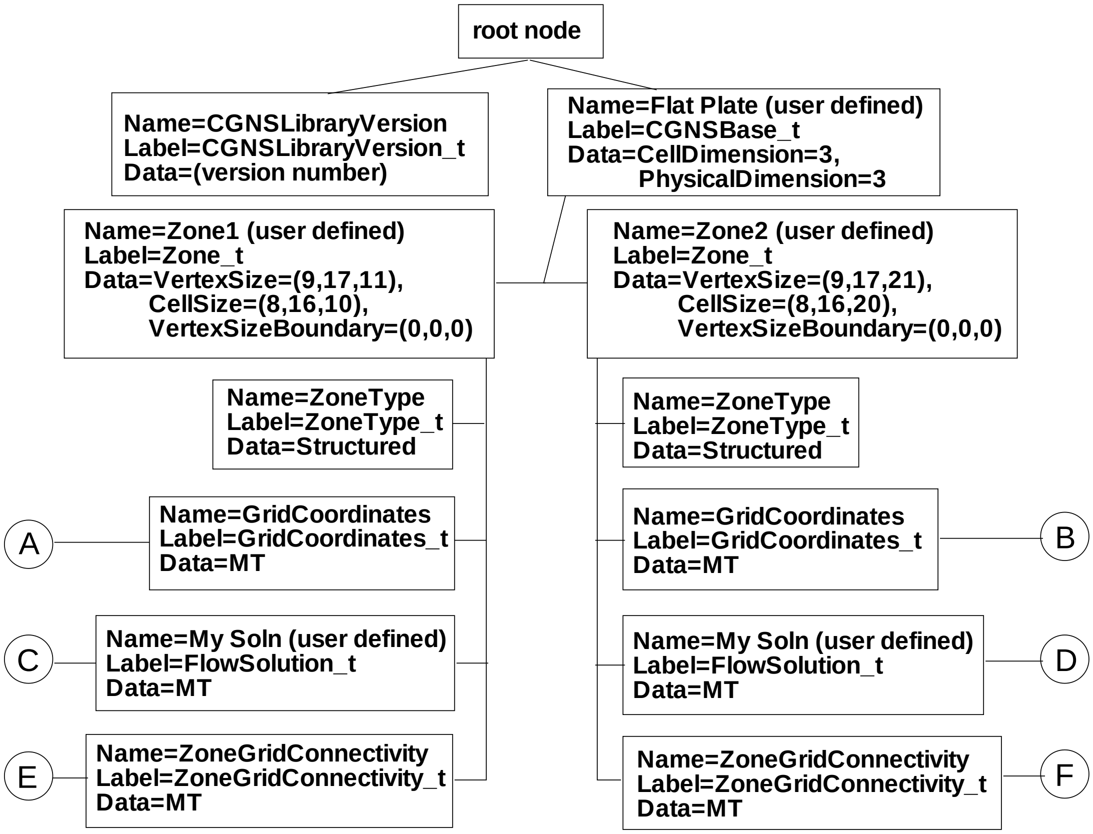

Figure 27: CGNS top levels for a case composed of 2 structured zones.

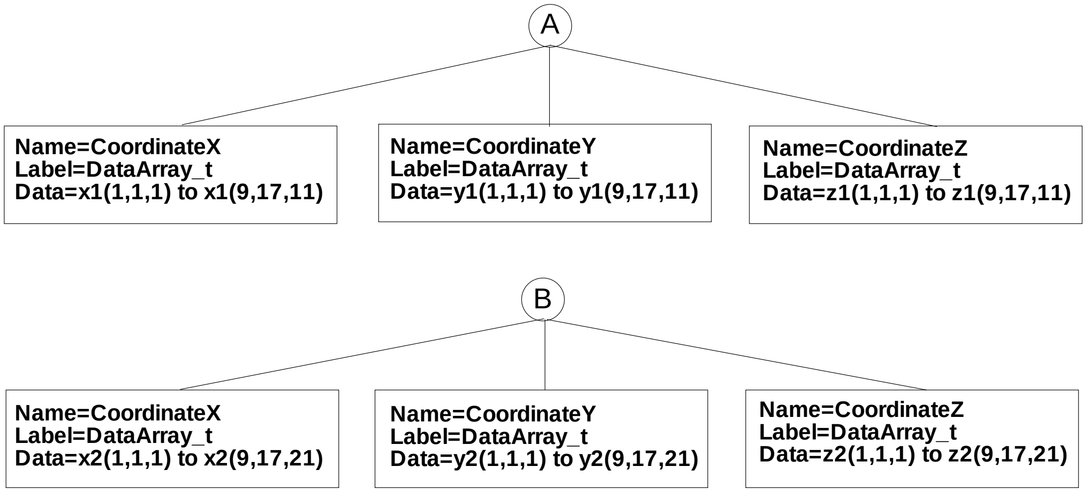

Figure 28: `GridCoordinate_t` nodes of structured zone example.

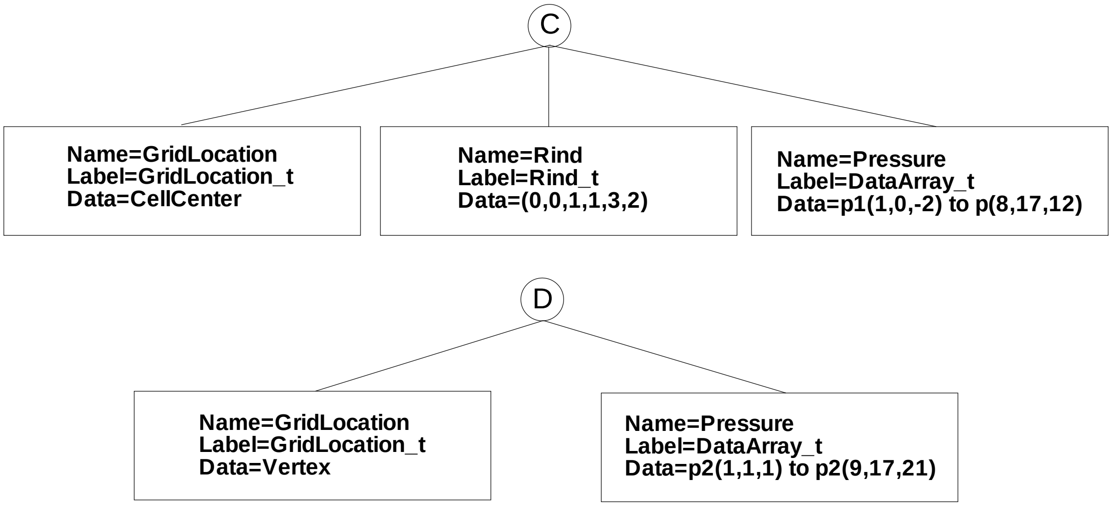

Figure 29: FlowSolution_t nodes of structured zone example.

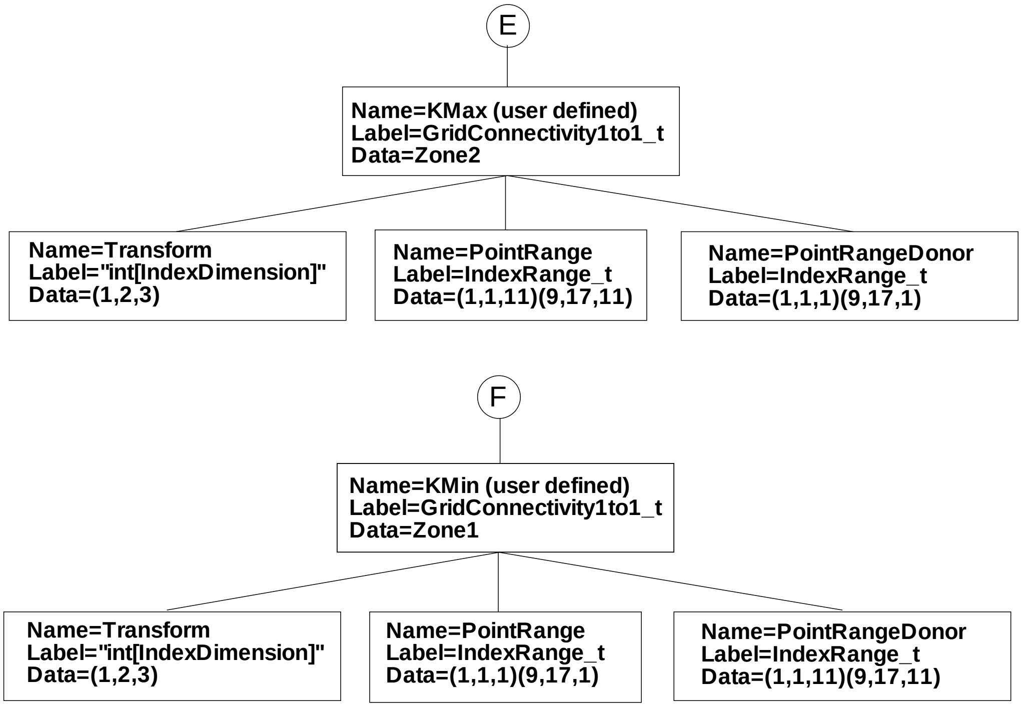

Figure 30: ZoneGridConnectivity_t nodes of structured zone example.

boundaries of the zone. (A rind node can also be used under GridCoordinates_t and DiscreteData_t.) See the SIDS document [1] for a more complete description. In zone 1 in this example, there are no additional ghost cell data in the i-direction, there is one ghost cell next to each of j-min and j-max, and there are 3 ghost cells next to k-min and 2 next to k-max. (Admittedly, this example is very contrived - most applications would be more consistent in their use of rind cells.) Because of the rind cells, the i, j, and k ranges of all flow solution data arrays in zone 1 are extended appropriately.

It is very important for the user to realize that including rind cells affects how the data is stored in the DataArray_t's. In other words, when reading a CGNS file one cannot ignore Rind_t nodes if they are present, and attempting to read the DataArray_t's using unmodified VertexSize or CellSize dimensions will result in the retrieval of nonsensical data.

Note that the SIDS specifies many defaults. For example, the default Transform values are (1,2,3), and the default GridLocation is Vertex. Hence, the nodes that contain these particular values in the example are not strictly necessary. The API sometimes leaves out default information.

Another important fact is illustrated in this example. When the names of a type of node (of given label) are user defined, the names must be *different* if they have the same parent node. For example, the two Zone_t nodes in this example must have different names (recall the earlier discussion of zone naming). However, if they are located in different places in the hierarchy, two nodes with the same label can have the same name. For example, both of the FlowSolution_t nodes, located in two different zones, have been given the same user-defined name: "My Soln" in the example.

Finally, although the ZoneBC_t nodes were not included in this example, note that if they were, they should describe the boundary conditions on all boundary faces *except* the k-max face of zone 1 and the k-min face of zone 2. These two faces would not be included in the boundary conditions because they are already defined as connectivity interfaces.

Appendix C. GUIDELINE FOR PLOT3D VARIABLES

The broad scope of CGNS allows users to essentially put *anything* into a CGNS file. While this is useful from the perspective of extensibility, it also makes it more difficult to read someone else's CGNS file without an elaborate array of checks and translators. This is true not only because of the many choices of variables to output, but also because CGNS allows many forms of dimensional and nondimensional data.

Many people in the CFD community currently output structured-grids and corresponding flow field data in PLOT3D format [7], particularly for use in postprocessing visualization programs. It has, in some sense, become a *de facto* standard for sharing structured CFD data. Because this format is so widely used, we give a guideline in this appendix for outputting and reading this type of data in a CGNS file. If you follow this guideline, then it is more likely that other users will be able to easily read and interpret your CGNS files.

The PLOT3D standard grid variables are (in 3-D) x, y, and z. These coordinates may be dimensional or nondimensional. To follow this guideline, the three coordinates `CoordinateX`, `CoordinateY`, and `CoordinateZ` (either dimensional or nondimensional) must be given. There also may be "iblank" information, associated with overset grids. If used, the list of overset holes is stored under `OversetHoles_t` nodes (see the SIDS document [1]). This appendix does not cover the various dimensionalization and nondimensionalization options for the grid coordinates. By and large, from the point of view of portability, the issue of units and/or nondimensionalization for grid coordinates is not as crucial as it is for the "Q" variables, which is covered in great detail below. However, one should follow the SIDS standard and appropriately define within the CGNS file the grid's units or nondimensionalizations used.

The PLOT3D standard "Q" variables are (in 3-D):

ρ/ρ_{ref} = nondimensional density

$\rho u/(\rho_{ref} a_{ref})$ = nondimensional x-momentum

$\rho v/(\rho_{ref} a_{ref})$ = nondimensional y-momentum

$\rho w/(\rho_{ref} a_{ref})$ = nondimensional z-momentum

$\rho e_0/(\rho_{ref} a_{ref}^2)$ = nondimensional total energy per unit volume

where a is the speed of sound and *ref* indicates a reference state. Standard PLOT3D Q files also specify a reference Mach number, Reynolds number, angle of attack, and time value. For the purposes of this discussion, the time value will not be addressed. CGNS does have the capability for storing time-accurate data if needed (see section 3.6), but time-accurate data is not covered in this PLOT3D guideline. We include below the CGNS convention for storing Mach number, Reynolds number, and (indirectly) angle of attack.

Each of the 5 flow field variables above has a standard name, defined in the SIDS. They are, respectively: `Density`, `MomentumX`, `MomentumY`, `MomentumZ`, and `EnergyStagnationDensity`. To follow this guideline, these are the 5 variables that should be output to your CGNS file (in 3-D), and are also the ones that you should expect to read, given someone else's CGNS file, if they are following this guideline.

Multiple bases are allowed in CGNS, but, to further enhance portability of PLOT3D-like datasets, only one `CGNSBase_t` node is recommended under this guideline. In other words, multiple cases (such as different angles of attack) should be stored in separate CGNS files with single bases, rather than in a single file with multiple bases.

The three most common types of data that one may output in a CGNS file are:

`DataClass = Dimensional`

`DataClass = NormalizedByDimensional`

`DataClass = NormalizedByUnknownDimensional`

The first category indicates that the data has dimensional units. The second category indicates that the data has been nondimensionalized by *known* reference values, which are specified in the CGNS file. The third category indicates that the data is nondimensional, but the reference values are unspecified or unknown. Because CGNS deals with each of these in a slightly different way, we will give the guideline for each of these three classes in separate subsections.

C.1 Dimensional Data

To output dimensional data:

1. Under `CGNSBase_t`, set `DataClass = Dimensional`.
2. Under `CGNSBase_t`, put a `ReferenceState`; and under `ReferenceState`, put the dimensional reference values of `Density` and `VelocitySound`. Under this guideline, the units of these must be consistent with one another and with the units of `Density`, `MomentumX`, `MomentumY`, `MomentumZ`, and `EnergyStagnationDensity` given under `FlowSolution` (e.g., all MKS units). Also under `ReferenceState`, put `Mach`, `Reynolds`, `VelocityX`, `VelocityY`, and `VelocityZ`.
3. Under `FlowSolution`, put the dimensional variables `Density`, `MomentumX`, `MomentumY`, `MomentumZ`, and `EnergyStagnationDensity`. Under this guideline, the units of these 5 variables must be consistent with one another and with the units of `Density` and `VelocitySound` in `ReferenceState`.

To read dimensional data (i.e., if `DataClass = Dimensional` under `CGNSBase_t`):

1. Under `ReferenceState` (directly under `CGNSBase_t`), read `Density`, `VelocitySound`, `Mach`, and `Reynolds`. Also read `VelocityX`, `VelocityY`, and `VelocityZ` if an angle of attack of the reference state is needed.
2. Under `FlowSolution`, read `Density`, `MomentumX`, `MomentumY`, `MomentumZ`, and `EnergyStagnationDensity`.

3. To obtain the PLOT3D Q variables, do the following:

   ρ/ρ_{ref} = Density / Density(ref)

   $\rho u/(\rho_{ref} a_{ref})$ = MomentumX / (Density(ref) * VelocitySound(ref))

   $\rho v/(\rho_{ref} a_{ref})$ = MomentumY / (Density(ref) * VelocitySound(ref))

   $\rho w/(\rho_{ref} a_{ref})$ = MomentumZ / (Density(ref) * VelocitySound(ref))

   $\rho e_0/(\rho_{ref} a_{ref}^2)$ = EnergyStagnationDensity / (Density(ref) * VelocitySound(ref)^2)

C.2 NormalizedByDimensional Data

To output nondimensional data with known normalizations:

1. Under CGNSBase_t, set DataClass = NormalizedByDimensional.

2. Under CGNSBase_t, put a ReferenceState; and under ReferenceState, put Mach, Reynolds, VelocityX, VelocityY, and VelocityZ. Then put either:

   - The *dimensional* reference values of Density and VelocitySound. Under this guideline, the units of these must be consistent with one another and with the units of the *raw* (dimensional) data Density, MomentumX, MomentumY, MomentumZ, and EnergyStagnationDensity given under FlowSolution, prior to normalization.
   - The *nondimensional* reference values of Density and VelocitySound, along with their corresponding ConversionScale and ConversionOffset values. Under this guideline, the units of the *raw* (dimensional) Density and VelocitySound, prior to normalization using ConversionScale and ConversionOffset, must be consistent with one another and with the units of the *raw* (dimensional) data Density, MomentumX, MomentumY, MomentumZ, and EnergyStagnationDensity given under FlowSolution, prior to normalization.

3. Under FlowSolution, put the nondimensional variables Density, MomentumX, MomentumY, MomentumZ, and EnergyStagnationDensity, along with their corresponding ConversionScale and ConversionOffset values. Under this guideline, the units of the *raw* (dimensional) variables, prior to normalization using ConversionScale and ConversionOffset, must be consistent with one another and with the units of the *raw* (dimensional) Density and VelocitySound in ReferenceState.

To read nondimensional data with known normalizations (i.e., if DataClass = NormalizedByDimensional under CGNSBase_t):

1. Under ReferenceState (directly under CGNSBase_t), read Density and VelocitySound. Also read their ConversionScale and ConversionOffset values if they are present. Additionally, read Mach and Reynolds. Also read VelocityX, VelocityY, and VelocityZ if an angle of attack of the reference state is needed.

2. Under FlowSolution, read Density, MomentumX, MomentumY, MomentumZ, and EnergyStagnationDensity. Also read each ConversionScale and ConversionOffset.

3. To obtain the PLOT3D Q variables, do the following. First, *only if they were given as nondimensional quantities* (indicated by a ' below), recover the *raw* (dimensional) reference values of Density and VelocitySound, via:

   Density(ref) = Density'(ref)*ConversionScale + ConversionOffset

   VelocitySound(ref) = VelocitySound'(ref)*ConversionScale + ConversionOffset

   Then do:

   ρ/ρ_{ref} = (Density*ConversionScale + ConversionOffset) / Density(ref)

   $\rho u/(\rho_{ref} a_{ref})$ = (MomentumX*ConversionScale + ConversionOffset) / (Density(ref) * VelocitySound(ref))

   $\rho v/(\rho_{ref} a_{ref})$ = (MomentumY*ConversionScale + ConversionOffset) / (Density(ref) * VelocitySound(ref))

   $\rho w/(\rho_{ref} a_{ref})$ = (MomentumZ*ConversionScale + ConversionOffset) / (Density(ref) * VelocitySound(ref))

   $\rho e_0/(\rho_{ref} a_{ref}^2)$ = (EnergyStagnationDensity*ConversionScale + ConversionOffset) / (Density(ref) * VelocitySound(ref)^2)

Note that it is possible that the conversion scale and offset for the PLOT3D Q variables may correspond to the reference conditions. This would imply that the variables could be directly output, without the above conversions needed. However, CGNS allows the variables to be normalized by properties independent of the reference conditions, so the above procedure is recommended to avoid ambiguity.

C.3 NormalizedByUnknownDimensional Data

To output nondimensional data with unknown normalizations:

1. Under CGNSBase_t, set DataClass = NormalizedByUnknownDimensional.

2. Under CGNSBase_t, put a ReferenceState; and under ReferenceState, put Density = 1 and VelocitySound = 1. Also, put Mach, Reynolds, VelocityX, VelocityY, and VelocityZ.

3. Under FlowSolution, put the nondimensional variables Density, MomentumX, MomentumY, MomentumZ, and EnergyStagnationDensity. These must be nondimensionalized as: ρ/ρ_{ref}, $\rho u/(\rho_{ref} a_{ref})$, $\rho v/(\rho_{ref} a_{ref})$, $\rho w/(\rho_{ref} a_{ref})$, $\rho e_0/(\rho_{ref} a_{ref}^2)$.

(Setting Density = 1 and VelocitySound = 1 in the ReferenceState defines the particular nondimensionalization defined above for the PLOT3D variables; see the SIDS document [1] for details and other examples.) To read nondimensional data with unknown normalizations (i.e., if DataClass = NormalizedByUnknownDimensional under CGNSBase_t):

1. Check under ReferenceState (directly under CGNSBase_t), to be sure that Density = 1 and VelocitySound = 1. Then, read Mach and Reynolds. Also read VelocityX, VelocityY, and VelocityZ if an angle of attack of the reference state is needed.

2. Under FlowSolution, read Density, MomentumX, MomentumY, MomentumZ, and EnergyStagnationDensity.

Nothing needs to be done in this case to obtain the PLOT3D Q variables. They are already in the correct form.

C.4 Notes

1. In addition to the flow field variables Density, MomentumX, MomentumY, MomentumZ, and EnergyStagnationDensity (under FlowSolution), you may also output additional variables if desired, but be sure these 5 are present.

2. Other reference values may also be placed under ReferenceState (for example, LengthReference may be needed to define the reference length associated with the grid coordinates), but the use of Density and VelocitySound is sufficient to define the nondimensionalizations of the PLOT3D Q variables.

3. The quantities Mach, Reynolds, VelocityX, VelocityY, VelocityZ, Density, and VelocitySound (plus anything else) under ReferenceState must all represent the same reference state of the flow. For external aerodynamics, this is usually taken to be the free stream, but it does not have to be.

4. The velocity components are used, in the PLOT3D sense, solely to provide an angle of attack of the flow field at the reference state. The definition of angle of attack itself is non-unique in 3-D, so there is therefore no SIDS standard for it. For example, one possible set of angle definitions assumes that the z-direction is "up," and uses:

   $u = V \cos\beta \cos\alpha$

   $v = -V \sin\beta$

   $w = V \cos\beta \sin\alpha$

   where $V = \sqrt{u^2 + v^2 + w^2}$, α is angle of attack, and β is angle of sideslip. Thus, an angle of attack can be obtained using $\alpha = \tan^{-1}(w/u)$, where u = VelocityX and w = VelocityZ.

5. When reading someone else's CGNS file, a low-level approach to interpret and/or use it appropriately would be the following. First, check to see that there is only one CGNSBase_t node. (As discussed above, multiple bases are allowed in general, but under this guideline only one base should exist.) Second, insure that the variables CoordinateX, CoordinateY, and CoordinateZ exist under each zone's GridCoordinates_t node, and that the variables Density, MomentumX, MomentumY, MomentumZ, and EnergyStagnationDensity exist under each zone's FlowSolution_t node. (Note: for time-accurate datasets there may be multiple GridCoordinates_t and FlowSolution_t nodes under each zone – see section 3.6 – but this situation is not covered under the current PLOT3D guideline.) Then, search for the following characteristics in the file:

   - If DataClass = Dimensional, then ReferenceState (directly under CGNSBase_t) *must* contain Density, VelocitySound, Mach, and Reynolds. VelocityX, VelocityY, and VelocityZ are needed under ReferenceState only if a reference angle of attack is required.
   - If DataClass = NormalizedByDimensional, then ReferenceState (directly under CGNSBase_t) *must* contain Density, VelocitySound, Mach, and Reynolds. VelocityX, VelocityY, and VelocityZ are needed under ReferenceState only if a reference angle of attack is required. Furthermore, a ConversionScale and ConversionOffset must exist for each of the 5 flow field variables under FlowSolution. ConversionScale and ConversionOffset may or may not exist for the variables under ReferenceState.
   - If DataClass = NormalizedByUnknownDimensional, then ReferenceState (directly under CGNSBase_t) *must* contain Density = 1, VelocitySound = 1, as well as Mach, and Reynolds. VelocityX, VelocityY, and VelocityZ are needed under ReferenceState only if a reference angle of attack is required.

   If these conditions are met, then a low-level reader could assume that the guidelines outlined in the above subsections were followed, and the PLOT3D variables could easily be obtained using the procedures given. A more advanced reader would probably check for consistency in the dimensions and conversion scales, to ensure compliance with the guidelines.

References

[1] CGNS Project Group, "The CFD General Notation System Standard Interface Data Structures," Version 2.0 beta 2, February 2001; http://www.grc.nasa.gov/www/cgns/sids/index.html

[2] CGNS Project Group, "The CFD General Notation System Advanced Data Format (ADF) User's Guide," April 2001; http://www.grc.nasa.gov/www/cgns/adf/index.html

[3] CGNS Project Group, “The CFD General Notation System SIDS-to-ADF File Mapping Manual,” Version 1.2 revision 8, February 2001; http://www.grc.nasa.gov/www/cgns/filemap/index.html

[4] Poirier, D. M. A., Allmaras, S., McCarthy, D. R., Smith, M., and Enomoto, F., “The CGNS System,” AIAA Paper 98-3007, June 1998.

[5] CGNS Project Group, “The CFD General Notation System Mid-Level Library,” July 2001; http://www.grc.nasa.gov/www/cgns/midlevel/index.html

[6] 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, January 2000.

[7] Walatka, P. P., Buning, P. G., Pierce, L., Elson, P. A., “PLOT3D User’s Guide,” NASA TM 101067, March 1990.