

Recommended Practice

The CFD General Notation System – Standard Interface Data Structures

Recommended Practice

The CFD General Notation System – Standard Interface Data Structures

Maintained by
The CGNS Steering Sub-committee of the AIAA CFD Committee on Standards

Abstract

The CFD General Notation System (CGNS) is a standard for recording and recovering computer data associated with the numerical solution of the equations of fluid dynamics. The intent is to facilitate the exchange of CFD data between sites, between applications codes, and across computing platforms, and to stabilize the archiving of CFD data.

The CGNS system consists of a collection of conventions, and software implementing those conventions, for the storage and retrieval of CFD data. It consists of two parts: (1) a standard format for recording the data, and (2) software that reads, writes, and modifies data in that format. The format is a conceptual entity established by the documentation; the software is a physical product supplied to enable developers to access and produce data recorded in that format.

The standard format, or paper convention, part of CGNS consists of two fundamental pieces. The first, known as the Standard Interface Data Structures, is described in this Recommended Practice. It defines the intellectual content of the information to be stored. The second, known as the File Mapping, defines the exact location in a CGNS file where the data is to be stored.

Published by

American Institute of Aeronautics and Astronautics
1801 Alexander Bell Drive, Reston, VA 20191

Copyright © 2007 American Institute of Aeronautics and
Astronautics
All rights reserved.

No part of this publication may be reproduced in any form, in an electronic
retrieval system or otherwise, without prior written permission of the publisher.

Printed in the United States of America.

Contents

Foreword	vii
1 Introduction	1
1.1 Major Differences from Previous CGNS Versions	2
1.1.1 Version 2.0, Beta 1	3
1.1.2 Version 2.0, Beta 2	3
1.1.3 Version 2.1, Beta 1	3
1.1.4 Version 2.2, Beta 1	4
1.1.5 Version 2.3	4
1.1.6 Version 2.4	4
2 Design Philosophy of Standard Interface Data Structures	6
3 Conventions	9
3.1 Data Structure Notation Conventions	9
3.2 Structured Grid Notation and Indexing Conventions	13
3.3 Unstructured Grid Element Numbering Conventions	14
3.3.1 1-D (Line) Elements	15
3.3.2 2-D (Surface) Elements	16
3.3.2.1 Triangular Elements	16
3.3.2.2 Quadrilateral Elements	17
3.3.3 3-D (Volume) Elements	17
3.3.3.1 Tetrahedral Elements	17
3.3.3.2 Pyramid Elements	18
3.3.3.3 Pentahedral Elements	20
3.3.3.4 Hexahedral Elements	21
3.3.4 Unstructured Grid Example	22
3.4 Multizone Interfaces	24
4 Building-Block Structure Definitions	27
4.1 Definition: DataClass_t	27
4.2 Definition: Descriptor_t	27
4.3 Definition: DimensionalUnits_t	28
4.4 Definition: DimensionalExponents_t	29
4.5 Definition: GridLocation_t	30
4.6 Definition: IndexArray_t	30
4.7 Definition: IndexRange_t	30
4.8 Definition: Rind_t	31
5 Data-Array Structure Definitions	32
5.1 Definition: DataArray_t	32
5.1.1 Definition: DataConversion_t	33
5.2 Data Manipulation	34
5.2.1 Dimensional Data	34

5.2.2	Nondimensional Data Normalized by Dimensional Quantities	35
5.2.3	Nondimensional Data Normalized by Unknown Dimensional Quantities	35
5.2.4	Nondimensional Parameters	38
5.2.5	Dimensionless Constants	39
5.3	Data-Array Examples	39
6	Hierarchical Structures	44
6.1	CGNS Version	44
6.2	CGNS Entry Level Structure Definition: CGNSBase_t	44
6.3	Zone Structure Definition: Zone_t	47
6.4	Precedence Rules and Scope Within the Hierarchy	51
7	Grid Coordinates, Elements, and Flow Solutions	53
7.1	Grid Coordinates Structure Definition: GridCoordinates_t	53
7.2	Grid Coordinates Examples	55
7.3	Elements Structure Definition: Elements_t	58
7.4	Elements Examples	60
7.5	Axisymmetry Structure Definition: Axisymmetry_t	61
7.6	Rotating Coordinates Structure Definition: RotatingCoordinates_t	63
7.7	Flow Solution Structure Definition: FlowSolution_t	64
7.8	Flow Solution Example	66
8	Multizone Interface Connectivity	69
8.1	Zonal Connectivity Structure Definition: ZoneGridConnectivity_t	69
8.2	1-to-1 Interface Connectivity Structure Definition: GridConnectivity1to1_t	70
8.3	1-to-1 Interface Connectivity Examples	73
8.4	General Interface Connectivity Structure Definition: GridConnectivity_t	75
8.5	Grid Connectivity Property Structure Definition: GridConnectivityProperty_t	78
8.5.1	Periodic Interface Structure Definition: Periodic_t	79
8.5.2	Average Interface Structure Definition: AverageInterface_t	80
8.6	Overset Grid Holes Structure Definition: OversetHoles_t	81
9	Boundary Conditions	83
9.1	Boundary Condition Structures Overview	84
9.2	Zonal Boundary Condition Structure Definition: ZoneBC_t	85
9.3	Boundary Condition Structure Definition: BC_t	86
9.4	Boundary Condition Data Set Structure Definition: BCDataSet_t	90
9.5	Boundary Condition Data Structure Definition: BCData_t	92
9.6	Boundary Condition Property Structure Definition: BCProperty_t	93
9.6.1	Wall Function Structure Definition: WallFunction_t	94
9.6.2	Area Structure Definition: Area_t	95
9.7	Boundary Condition Type Structure Definition: BCType_t	96
9.8	Matching Boundary Condition Data Sets	99
9.9	Boundary Condition Specification Data	101
9.10	Boundary Condition Examples	103

10 Governing Flow Equations	112
10.1 Flow Equation Set Structure Definition: FlowEquationSet_t	112
10.2 Governing Equations Structure Definition: GoverningEquations_t	113
10.3 Thermodynamic Gas Model Structure Definition: GasModel_t	115
10.4 Molecular Viscosity Model Structure Definition: ViscosityModel_t	116
10.5 Thermal Conductivity Model Structure Definition: ThermalConductivityModel_t	118
10.6 Turbulence Structure Definitions	119
10.6.1 Turbulence Closure Structure Definition: TurbulenceClosure_t	120
10.6.2 Turbulence Model Structure Definition: TurbulenceModel_t	121
10.7 Thermal Relaxation Model Structure Definition: ThermalRelaxationModelType_t	124
10.8 Chemical Kinetics Structure Definition: ChemicalKineticsModel_t	125
10.9 Electromagnetics Structure Definitions	127
10.9.1 Electromagnetics Electric Field Model Structure Definition: EMElectricField- Model_t	127
10.9.2 Electromagnetics Magnetic Field Model Structure Definition: EMMagneticField- Model_t	128
10.9.3 Electromagnetics Conductivity Model Structure Definition: EMConductivity- Model_t	129
10.10 Flow Equation Examples	130
11 Time-Dependent Flow	133
11.1 Iterative Data Structure Definitions	133
11.1.1 Base Iterative Data Structure Definition: BaseIterativeData_t	133
11.1.2 Zone Iterative Data Structure Definition: ZoneIterativeData_t	134
11.2 Rigid Grid Motion Structure Definition: RigidGridMotion_t	135
11.3 Arbitrary Grid Motion Structure Definition: ArbitraryGridMotion_t	137
11.4 Examples for Time-Dependent Flow	140
12 Miscellaneous Data Structures	146
12.1 Reference State Structure Definition: ReferenceState_t	146
12.2 Reference State Example	147
12.3 Convergence History Structure Definition: ConvergenceHistory_t	148
12.4 Discrete Data Structure Definition: DiscreteData_t	150
12.5 Integral Data Structure Definition: IntegralData_t	151
12.6 Family Data Structure Definition: Family_t	152
12.7 Geometry Reference Structure Definition: GeometryReference_t	153
12.8 Family Boundary Condition Structure Definition: FamilyBC_t	154
12.9 User-Defined Data Structure Definition: UserDefinedData_t	155
12.10 Gravity Data Structure Definition: Gravity_t	156
Annex A. Conventions for Data-Name Identifiers	159
A.1 Coordinate Systems	159
A.2 Flowfield Solution	160
A.3 Turbulence Model Solution	168
A.4 Nondimensional Parameters	168

A.5 Characteristics and Riemann Invariants Based on 1-D Flow	170
A.6 Forces and Moments	170
A.7 Time-Dependent Flow	173

Annex B. Structured Two-Zone Flat Plate Example 175

B.1 Overall Layout	176
B.2 Grid Coordinates	177
B.3 Flowfield Solution	178
B.4 Interface Connectivity	180
B.5 Boundary Conditions	183
B.6 Global Reference State	186
B.7 Equation Description	188

List of Figures

1 Sample Topologically Based CFD Hierarchy	7
2 Structured-Grid Multizone Interface Types	25
3 Example Interface for 1-to-1 Connectivity	74
4 Hierarchy for Boundary Condition Structures	85
5 Boundary Condition Implementation Levels	103
6 Two-Zone Flat Plate Test Case	175

List of Tables

1 Element Types in CGNS	15
2 Simple Boundary Condition Types	97
3 Compound Boundary Condition Types	99
4 Associated Boundary Condition Types and Usage Rules	100
5 Data-Name Identifiers for Perfect Gas	116
6 Data-Name Identifiers for Molecular Viscosity Models	118
7 Data-Name Identifiers for Thermal Conductivity Models	120
8 Data-Name Identifiers for Turbulence Closure	121
9 Data-Name Identifiers for Chemical Kinetics Models	126
10 Defined Names (Symbols) for Commonly Used Mixtures	126
11 Data-Name Identifiers for Electromagnetics Models	130
12 Data-Name Identifiers for Rigid Grid Motion	137
13 Data-Name Identifiers for Grid Velocity	139
14 Data-name Identifiers for Reference State	147
15 Data-Name Identifiers for Coordinate Systems	160
16 Data-Name Identifiers for Flow Solution Quantities	162
17 Data-Name Identifiers for Typical Turbulence Models	168
18 Data-Name Identifiers for Nondimensional Parameters	169
19 Data-Name Identifiers for Characteristics and Riemann Invariants	170
20 Data-Name Identifiers for Forces and Moments	172
21 Data-Name Identifiers for Time-Dependent Flow	173

Foreword

This document contains the Standard Interface Data Structures (SIDS) definitions for the CFD General Notation System (CGNS) project. This project was originally a NASA-funded contract under the AST program, but control has now been completely transferred to a public forum known as the CGNS Steering Committee, a sub-committee of the AIAA CFD Committee on Standards.

The purpose of this document is to scope the information that should be communicated between various CFD application codes; the target is 3-D multizone, compressible Navier-Stokes analysis. Attention in this document is not focused on I/O routines or formats, but on the precise description of data that should be present in the I/O of a CFD code or in a CFD database.

This document therefore contains a precise definition of information pertinent to a CGNS database. Specifically, the following information is addressed:

- grid coordinates and elements
- flow solution data, including nondimensional parameters
- multizone interface connectivity, including abutting and overset
- boundary conditions
- flow equation descriptions
- time-dependent flow
- reference states
- dimensional units and nondimensionalization information associated with data
- convergence history
- association to geometry definition
- topologically based hierarchical structures

This information is encoded into C-like data structures.

This revision “A” of AIAA R-101 replaces in whole AIAA R-101-2002. Changes made to the document since its last release are outlined in Section 1. At the time of approval, the members of the CGNS Steering Committee were:

Chris Rumsey, Chairman	NASA Langley
Steve Allmaras	Boeing Commercial
Juan Alonso	Stanford University
Bob Bush	Pratt & Whitney
Thierry Chevalier	AM-Airbus
Armen D. Darian	Pratt & Whitney Rocketdyne
Dan Dominik	Boeing Rocketdyne
Steve Feldman	CD-ADAPCO

Pankaj Gupta	Fluent
Richard Hann	ANSYS / CFX
Thomas Hauser	Utah State
Steve Legensky	Intelligent Light
Todd Michal	Boeing Integrated Defense Systems
Marc Poinot	ONERA
Greg Power	ATA / AEDC
Charles Towne	NASA Glenn
Kurt Weber	Rolls Royce / Allison
Bruce Wedan	Ansys / ICEM CFD Engineering
Nick Wyman	Pointwise, Inc.

The above consensus body approved this document in December 2005.

The AIAA Standards Executive Council (Mr. Amr ElSawy, Chairman) accepted the document for publication in December 2005.

The AIAA Standards Procedures dictates that all approved Standards, Recommended Practices, and Guides are advisory only. Their use by anyone engaged in industry or trade is entirely voluntary. There is no agreement to adhere to any AIAA standards publication and no commitment to conform to or be guided by standards reports. In formulating, revising, and approving standards publications, the committees on standards will not consider patents that may apply to the subject matter. Prospective users of the publications are responsible for protecting themselves against liability for infringement of patents or copyright or both.

Questions and comments on this document are welcome and should be directed to:

Charlie Towne
MS 86-7
NASA Glenn Research Center
Cleveland, OH 44135-3191
(216) 433-5851
(216) 977-7500 (FAX)
e-mail: town@nasa.gov

1 Introduction

CGNS (CFD General Notation System) is a collection of conventions, along with software implementing those conventions, for the storage and retrieval of CFD (computational fluid dynamics) data. The CGNS system is designed to facilitate the exchange of data between sites and applications, as well as to help stabilize the archiving of fluid dynamic data. In today's environment, it is important in many technical arenas to maintain detailed records of scientific computations. CGNS was designed to help promote a long-lasting and extensible standard for this purpose. Many companies and institutions choose to adopt the CGNS standard in order to increase productivity by (1) reducing the time required to translate between data created and used by different applications, and (2) increasing the quality, longevity, and re-usability of archived data.

The CGNS standard is a conceptual entity established by the documentation. The CGNS software is a physical product supplied to enable writing and reading data according to this standard. All CGNS software is completely free and open to anyone. By using the supplied software, it is relatively easy for users to adhere to most of the standard described in detail in this document.

The CGNS project originated during 1994 through a series of meetings that addressed improved transfer of NASA technology to industry. A principal impediment in this process was the disparity in I/O formats employed by various flow codes, grid generators, and other utilities, and CGNS was conceived as a means to promote "plug-and-play" CFD. Agreement was reached to develop CGNS at Boeing, under NASA Contract NAS1-20267, with active participation by a team of CFD researchers from NASA's Langley, Lewis (now Glenn), and Ames Research Centers, McDonnell Douglas Corporation (now part of Boeing), and Boeing Commercial Airplane Group. This team, which was joined by ICEM CFD Engineering Corporation of Berkeley, California in 1997, undertook the core of the development. However, in the spirit of creating a completely open and broadly accepted standard, all interested parties were encouraged to participate; the US Air Force and Arnold Engineering Development Center were notably present. From the beginning, the purpose was to develop a system that could be distributed freely, including all documentation, software and source code. This goal has now been fully realized; further, control of CGNS has been completely transferred to a public forum known as the CGNS Steering Committee.

The principal target is the data normally associated with compressible viscous flow (i.e., the Navier-Stokes equations), but the standard is also applicable to subclasses such as Euler and potential flows. The initial release addressed multi-zone grids, flow fields, boundary conditions, and zone-to-zone connection information, as well as a number of auxiliary items, such as non-dimensionalization, reference states, and equation set specifications. Extensions incorporated since then include unstructured mesh, connections to geometry definition, time-dependent flow, and support for multiple species and chemistry.

It is worth noting that extensibility is a fundamental design characteristic of the system, which in principal could be used for other disciplines of computational field physics, such as acoustics or electromagnetics, given the willingness of the cognizant scientific community to define the conventions.

The standard format, or paper convention, part of CGNS consists of two fundamental pieces. The first, known as the Standard Interface Data Structures (SIDS), describes in detail the intellectual content of the information to be stored. It defines, for example, the precise meaning of a "boundary

condition". The second, known as the File Mapping (either SIDS to ADF, or SIDS to HDF, depending on the underlying database manager being used), defines the exact location in a CGNS file where the data is to be stored.

The implementation, or software, part of CGNS likewise consists of two separate entities. CGNS files are read and written by a stand-alone database manager, either ADF (Advanced Data Format) or HDF (Hierarchical Data Format). The database manager implements a tree-like data structure, as a binary file. Since the format of this file is completely controlled by the database manager, and since ADF and HDF are both written in ANSI C (Fortran wrappers are provided), these files and the database manager itself are portable to any environment which supports ANSI C. Both ADF and HDF are available separately and constitute useful tools for the storage of large quantities of scientific data.

The underlying database manager, however, implements no knowledge of CFD or of the File Mapping. To simplify access to CGNS files, a second layer of software known as the Mid-Level Library is provided. This layer is in effect an API, or Application Programming Interface for CFD. The API incorporates knowledge of the CFD data structures, their meaning and their location in the file, enabling applications such as flow codes and grid generators to access the data in familiar terms. The API is therefore the piece of the CGNS system most visible to applications developers. Like the ADF and HDF database managers, the Mid-Level Library is written in ANSI C; all public API routines have Fortran counterparts.

This document presents the formal definition of the Standard Interface Data Structures (SIDS). Section 2 presents the major design philosophies used to develop the CGNS database and the encoding of this database into the SIDS; this section also provides an overview of the database hierarchy. Section 3 describes the C-like nomenclature conventions used to define the SIDS. This section also gives the conventions for structured grid indexing and unstructured element numbering, and the nomenclature for multizone interfaces. Low-level building-block structures are defined in Section 4; these structures are used to define all higher-level structures. Structures for defining data arrays, including dimensional-units and nondimensional information, are presented in Section 5. The top levels of the CGNS hierarchy are next defined in Section 6. The following sections then fill out the remainder of the hierarchy: Section 7 defines the grid-coordinate, elements, and flow-solution structures; Section 8 defines the multizone interface connectivity structures; Section 9 defines boundary-condition structures; Section 10 defines structures for describing governing flow equations; Section 11 defines structures related to time-dependent flows; and Section 12 contains miscellaneous structures. Two appendices complete the document. Annex A provides naming conventions for data contained within the CGNS database, and Annex B contains a complete SIDS description of a structured-grid two-zone test case.

1.1 Major Differences from Previous CGNS Versions

The following items represent noteworthy alterations and additions to the SIDS starting with the August 1999 draft document. (Note that some of these changes — notably those for unstructured zones, family, and geometry reference — have existed previously in separate documents, but are now being merged officially into the SIDS; the data structures themselves are not “new.”)

1.1.1 Version 2.0, Beta 1

The following changes were made for Version 2.0, Beta 1.

- The capability for recording unstructured zones has been added to the SIDS. (These changes occur throughout the document, although some specific items are listed below.)
- The values `UserDefined` and `Null` are now allowed for all enumeration types (throughout document).
- The following nodes are now defined (some of these also include additional new children sub-nodes): `Family_t` (Section 12.6), `Elements_t` (Section 7.3), `ZoneType_t` (Section 6.3), `FamilyName_t` (Section 6.3), `GeometryReference_t` (Section 12.7), `FamilyBC_t` (Section 12.8).
- Under `CGNSBase_t`, the `IndexDimension` is no longer recorded; it has been replaced by `CellDimension` and `PhysicalDimension` (Section 6.2).
- Under `Zone_t`, the optional parameter `VertexSizeBoundary` has been added for unstructured zones (Section 6.3).
- The method for general connectivity (`GridConnectivity_t`) has been altered. It now requires the use of either (a) `PointListDonor` (an integer, for `Abutting1to1` only) or (b) `CellListDonor` (an integer) plus `InterpolantsDonor` (a real) (Section 8.4).
- The `GridLocation_t` parameter has been moved up one level (from `BCDataSet_t` to `BC_t`). Thus, for example, if the boundary conditions are defined at vertices (the default), then any associated dataset information must also be specified at vertices (Section 9.3 and Section 9.4).
- The data-name identifier `LengthReference` has been added (Section 12.1 and Annex A.2).
- The ν_t parameter has been renamed `ViscosityEddyKinematic`, and a new parameter `ViscosityEddy`, representing μ_t , has been defined (Annex A.2).

1.1.2 Version 2.0, Beta 2

The following changes were made for Version 2.0, Beta 2.

- The following data structures related to time-dependent flow have been added: `BaseIterativeData_t` (Section 11.1.1), `ZoneIterativeData_t` (Section 11.1.2), `RigidGridMotion_t` (Section 11.2), `ArbitraryGridMotion_t` (Section 11.3).

1.1.3 Version 2.1, Beta 1

The following changes were made for Version 2.1, Beta 1.

- A node type `UserDefinedData_t` (Section 12.9) is added for the storage of arbitrary user defined data in `Descriptor_t` and `DataArray_t` children without the restrictions or implicit meanings imposed on these node types at other node locations.

- Support for multi-species flows and chemistry has been added. New gas models have been added to the `GasModelType_t` enumeration (Section 10.3), and `ThermalRelaxationModel_t` and `ChemicalKineticsModel_t` data structures have been added for describing the thermal relaxation and chemical kinetics models (Section 10.7 and Section 10.8). Additional flow solution data-name identifiers are included (Annex A.2).

1.1.4 Version 2.2, Beta 1

The following changes were made for Version 2.2, Beta 1.

- `Axisymmetry_t` and `RotatingCoordinates_t` nodes have been added, allowing the recording of data relevant to axisymmetric flows and rotating coordinates (Section 7.5 and Section 7.6).
- A `Gravity_t` node has been added for storage of the gravitational vector (Section 12.10).
- A `GridConnectivityProperty_t` node has been added, allowing the recording of special properties associated with particular connectivity patches, such as periodic interfaces, or interfaces where the data is to be averaged in some way prior to passing it to a neighboring interface (Section 8.5).
- A `BCProperty_t` node has been added, allowing the recording of special properties associated with particular boundary condition patches, such as wall function or bleed regions (Section 9.6).
- Additional flow solution data-name identifiers are included for variables in rotating coordinate systems (Annex A.2).

1.1.5 Version 2.3

The following changes were made for Version 2.3.

- `ElementRange` and `ElementList` have been added to the `BC_t` data structure. `ElementRange` or `ElementList` may now be used to define a boundary condition patch by specifying face indices, instead of using `PointRange` or `PointList` with `GridLocation` set to `FaceCenter`. The use of `PointRange` or `PointList` to define a boundary condition patch hasn't changed. They may be used to define a boundary condition patch by specifying either vertex or face indices.

When `PointRange` or `PointList` is used, the choice between vertex or face indices is determined by the value of `GridLocation_t`. When `ElementRange` or `ElementList` is used, `GridLocation_t` is ignored. (Section 9.3).

1.1.6 Version 2.4

The following changes were made for Version 2.4.

- `GridLocation_t`, `PointRange`, and `PointList` have been added to the `BCDataSet_t` data structure, allowing boundary conditions to be specified at locations different from those used to define the BC patch. (E.g., a BC patch may be defined using vertices, with boundary conditions applied at face centers.) (Section 9.4).
- Data structures have been added to `FlowEquationSet_t` for describing the electric field, magnetic field, and conductivity models used for electromagnetic flows. Corresponding recommended data-name identifiers have also been added. (Section 10.1 and Section 10.9).
- `RotatingCoordinates_t` has been added to the `Family_t` data structure. (Section 12.6).
- A `BCDataSet_t` list has been added to the `FamilyBC_t` data structure, allowing specification of boundary condition data arrays for CFD families. (Section 12.8).
- `GridLocation_t`, `PointRange`, `PointList`, `FamilyName_t`, `UserDefinedData_t`, and `Ordinal` have been added to the `UserDefinedData_t` data structure. (Section 12.9).
- The `DimensionalUnits_t` and `DimensionalExponents_t` structures have been expanded to include units for electric current, substance amount, and luminous intensity. (Section 4.3).

Downloaded by SHY AIAA Sobral (aiaa@aiaa.org) on 01/20/2015 14:43:03

The major design goal of the SIDS is a comprehensive and unambiguous description of the ‘intellectual content’ of information that must be passed from code to code in a multizone Navier-Stokes analysis system. This information includes grids, flow solutions, multizone interface connectivity, boundary conditions, reference states and dimensional units or normalization associated with data.

The goal is description of the data sets typical of CFD analysis, which tend to contain a small number of extremely large data arrays. This has a number of implications for both the design of the SIDS and the ultimate physical files where the data resides. The first is that any I/O system built for CFD analysis must be designed to efficiently store and process large data arrays. This is reflected in the SIDS, which includes provisions for describing large data arrays.

The third implication of CFD data sets is that files containing a CFD database are almost always required to be binary – ASCII storage of CFD data involves excessive storage and performance penalties. This means the files are not readable by humans and the information contained in them is not directly modifiable by text editors and such. This is reflected in the syntax of the SIDS, which tends to be verbose and thorough; whereas, directly modifiable ASCII file formats would tend to foster a more brief syntax.

It is important to note that the description of information by the SIDS is independent of physical file formats. However, it is targeted towards implementation using the ADF Core library. Some of the language components used to define the SIDS are meant to directly map into elements of an ADF node.

An early decision in the CGNS project was that any new CFD I/O standard should include a hierarchical database, such as a tree or directed graph. The SIDS describe a hierarchical database, precisely defining both the data and their hierarchical relationships.

There are two major alternatives to organizing a CFD hierarchy: topologically based and data-type based. In a topologically based graph, overall organization is by zones; information pertaining to a particular zone, including its grid coordinates or flow solution, hangs off the zone. In a data-type based graph, organization is by related data. For example, there would be two nodes at the same level, one for grid coordinates and another for the flow solution. Hanging off each of these nodes would be separate lists of the zones.

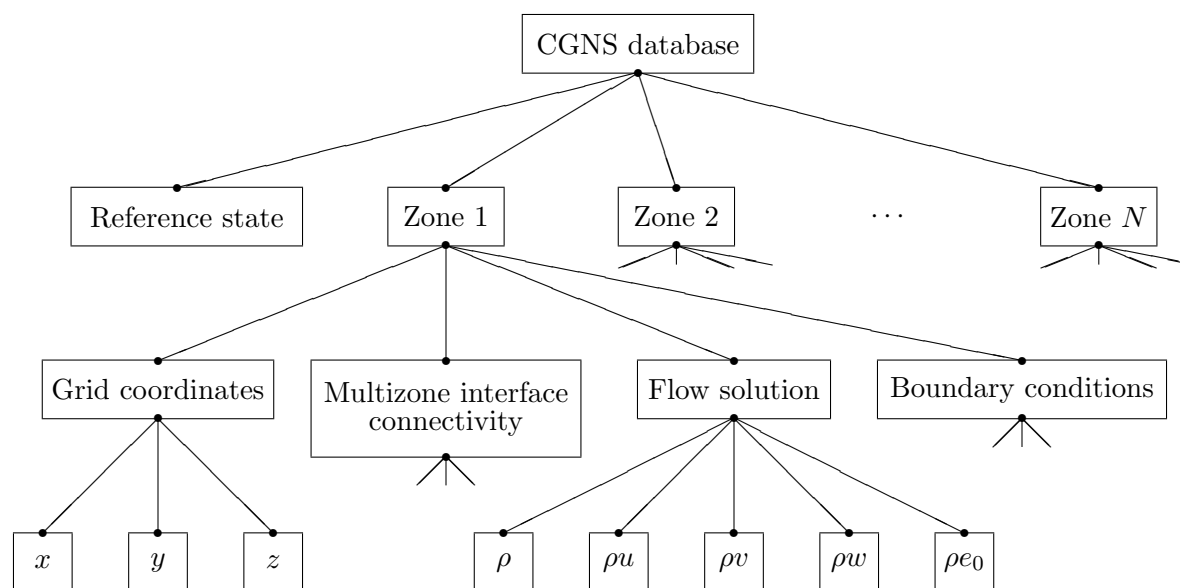


Figure 1: Sample Topologically Based CFD Hierarchy

The hierarchy described in this document is topologically based; a simplified illustration of the database hierarchy is shown in Figure 1. Hanging off the root ‘node’ of the database is a node containing global reference-state information, such as freestream conditions, and a list of nodes for each zone. The figure shows the nodes that hang off the first zone; similar nodes would hang off of each zone in the database. Nodes containing the physical-coordinate data arrays (x , y and z) for the first zone are shown hanging off the ‘grid coordinates’ node. Likewise, nodes containing the first zone’s flow-solution data arrays hang off the ‘flow solution’ node. The figure also depicts nodes containing multizone interface connectivity and boundary condition information for the first zone; subnodes hanging off each of these are not pictured.

Additional Design Objectives

The data structures comprising the SIDS are the result of several additional design objectives:

One objective is to minimize duplication of data within the hierarchy. Many parameters, such as the grid size of a zone, are defined in only one location. This avoids implementation problems arising from data duplication within the physical file containing the database; these problems include simultaneous update of all copies and error checking when two copies of a data quantity are found to be different. One consequence of minimizing data duplication is that information at lower levels of the hierarchy may not be completely decipherable without access to information at higher levels. For example, the grid size is defined in the zone structure (see Section 6.3), but this parameter is needed in several substructures to define the size of grid and flow-solution data arrays. Therefore, these substructures are not autonomous and deciphering information within them requires access to information contained in the zone structure itself. The SIDS must reflect this cascade of information within the database.

Another objective is elimination of nonsensical descriptions of the data. The SIDS have been

carefully developed to avoid data qualifiers and other optional descriptive information that could be inconsistent. This has led to the use of specialized structures for certain CFD-related information. One example is a single-purpose structure for defining physical grid coordinates of a zone. It is possible to define the grid coordinates, flow solution and any other field quantities within a zone by a generic discrete-data structure. However, this requires the generic structure to include information defining the grid location of the data (e.g. the data is located at vertices or cell centers). Using the generic structure to describe the grid coordinates leads to a possible inconsistency. By definition the physical coordinates that define the grid are located at vertices, and including an optional qualifier that states otherwise makes no sense.

A final objective is to allow documentation inclusion throughout the database. The SIDS contain a uniform documentation mechanism for all major structures in the hierarchy. However, this document establishes no conventions for using the documentation mechanism.

3 Conventions

3.1 Data Structure Notation Conventions

The intellectual content of the CGNS database is defined in terms of C-like notation including typedefs and structures. The database is made up of entities, and each entity has a type associated with it. Entities include such things as the dimensionality of the grid, an array of grid coordinates, or a zone which contains all the data associated with a given region. Entities are defined in terms of types, where a type can be an integer or a collection of elements (a structure) or a hierarchy of structures or other similar constructs.

The terminology ‘instance of an entity’ is used to refer to an entity that has been assigned a value or whose elements have been assigned values. The terminology ‘specific instance of a structure’ is also used in the following sections. It is short for an instance of an entity whose type is a structure.

Names of entities and types are constructed using conventions typical of *Mathematica*.¹ Names or identifiers contain no spaces and capitalization is used to distinguish individual words making up a name; names are case-sensitive. The characters ‘.’ and ‘/’ should be avoided in names as these have special meaning when referencing elements of a structure entity.

The following notational conventions are employed:

!	comment to end of line
_t	suffix used for naming a type
;	end of a definition, declaration, assignment or entity instance
=	assignment (takes on the value of)
:=	indicates a type definition (typedef)
[]	delimiters of an array
{ }	delimiters of a structure definition
{{ } }	delimiters of an instance of a structure entity
< >	delimiters of a structure parameter list
int	integer
real	floating-point number
char	character
bit	bit
Enumeration()	indicates an enumeration type
Data()	indicates an array of data, which may be multidimensional
List()	indicates a list of entities
Identifier()	indicates an entity identifier
LogicalLink()	indicates a logical link
/	delimiter for element of a structure entity
../	delimiter for parent of a structure entity
(r)	designation for a required structure element
(o)	designation for an optional structure element
(o/d)	designation for an optional structure element with default if absent

¹*Mathematica 3.0*, Wolfram Research, Inc., Champaign, IL (1996)

An enumeration type is a set of values identified by names; names of values within a given enumeration declaration must be unique. An example of an enumeration type is the following:

```
Enumeration( Dog, Cat, Bird, Frog )
```

This defines an enumeration type which contains four values.

Data() identifies an array of given dimensionality and size in each dimension, whose elements are all of a given data type. It is written as,

```
Data( DataType, Dimension, DimensionValues[] ) ;
```

Dimension is an integer, and **DimensionValues[]** is an array of integers of size **Dimension**. **Dimension** and **DimensionValues[]** specify the dimensionality of the array and its size in each dimension. **DataType** specifies the data type of the array's elements; it may consist of one of the following: **int**, **real**, **char** or **bit**. For multidimensional arrays, FORTRAN indexing conventions are used. **Data()** is formulated to map directly onto the data section of an ADF node.

A typedef establishes a new type and defines it in terms of previously defined types. Types are identified by the suffix '**_t**', and the symbol '**:=**' is used to establish a type definition (or typedef). For example, the above enumeration example can be used in a typedef:

```
Pet_t := Enumeration( Dog, Cat, Bird, Frog ) ;
```

This defines a new type **Pet_t**, which can then be used to declare a new entity, such as,

```
Pet_t MyFavoritePet ;
```

By the above typedef and declaration, **MyFavoritePet** is an entity of type **Pet_t** and can have the values **Dog**, **Cat**, **Bird** or **Frog**. A specific instance of **MyFavoritePet** is setting it equal to one of these values (e.g. **MyFavoritePet = Bird**).

A structure is a type that can contain any number of elements, including elements that are also structures. An example of a structure type definition is:

```
Sample_t :=
{
    int Dimension ;                                (r)

    real[4] Vector ;                               (o)

    Pet_t ObnoxiousPet ;                           (o)
} ;
```

where **Sample_t** is the type of the structure. This structure contains three elements, **Dimension**, **Vector** and **ObnoxiousPet**, whose types are **int**, **real[4]** and **Pet_t**, respectively. The type **int** specifies an integer, and **real[4]** specifies an array of reals that is one-dimensional with a length of four. The '**(r)**' and '**(o)**' notation in the right margin is explained below. Given the definition of **Sample_t**, entities of this type can then be declared (e.g. **Sample_t Sample1**);). An example of an instance of a structure entity is given by,

```

Sample_t Sample1 =
    {{
    Dimension = 3 ;
    Vector = [1.0, 3.45, 2.1, 5.4] ;
    ObnoxiousPet = Dog ;
    }} ;

```

Note the different functions played by single braces ‘{’ and double braces ‘{{’. The first is used to delimit the definition of a structure type; the second is used to delimit a specific instance of a structure entity.

Some structure type definitions contain arbitrarily long lists of other structures or types. These lists will be identified by the notation,

```
List( Sample_t Sample1 ... SampleN ) ;
```

where `Sample1 ... SampleN` is the list of structure names or identifiers, each of which has the type `Sample_t`. Within each list, the individual structure names are user-defined.

In the CGNS database it is sometimes necessary to reference the name or identifier of a structure entity. References to entities are denoted by `Identifier()`, whose single argument is a structure type. For example,

```
Identifier(Sample_t) SampleName ;
```

declares an entity, `SampleName`, whose value is the identifier of a structure entity of type `Sample_t`. Given this declaration, `SampleName` could be assigned the value `Sample1` (i.e. `SampleName = Sample1`).

It is sometimes convenient to directly identify an element of a specific structure entity. It is also convenient to indicate that two entities with different names are actually the same entity. We borrow UNIX conventions to indicate both these features, and make the analogy that a structure entity is a UNIX directory and its elements are UNIX files. An element of an entity is designated by ‘/’; an example is `Sample1/Vector`). The structure entity that a given element belongs to is designated ‘../’ A UNIX-like logical link that specifies the sameness of two apparently different entities is identified by `LogicalLink()`; it has one argument. An example of a logical link is as follows: Suppose a specific instance of a structure entity contains two elements that are of type `Sample_t`; call them `SampleA` and `SampleB`. The statement that `SampleB` is actually the same entity as `SampleA` is,

```
SampleB = LogicalLink(..../SampleA) ;
```

The argument of `LogicalLink()` is the UNIX-like ‘path name’ of the entity with which the link is made. In this document, `LogicalLink()` and the direct specification of a structure element via ‘/’ and ‘../’ are actually seldom used. These language elements are never used in the actual definition of a structure type.

Structure type definitions include three additional syntactic/semantic notions. These are parameterized structures, structure-related functions, and the identification of required and optional fields within a structure.

As previously stated, one of our design objectives is to minimize duplication of information within the CGNS database. To meet this objective, information is often stored in only one location of the hierarchy; however, that information is typically used in other parts of the hierarchy. A consequence of this is that it may not be possible to decipher all the information associated with a given entity in the hierarchy without knowledge of data contained in higher level entities. For example, the grid size of a zone is stored in one location (in `Zone_t`, see Section 6.3), but is needed in many substructures to define the size of grid and solution-data arrays.

This organization of information must be reflected in the language used to describe the database. First, parameterized structures are introduced to formalize the notion that information must be passed down the hierarchy. A given structure type is defined in terms of a list of parameters that precisely specify what information must be obtained from the structure's parent. These structure-defining parameters play a similar role to subroutine parameters in C or FORTRAN and are used to define fields within the structure; they are also passed onto substructures. Parameterized structures are also loosely tied to templates in C++.

Parameterized structures are identified by the delimiters `< >` enclosing the list of parameters. Each structure parameter in a structure-type definition consists of a type and an identifier. Examples of parameterized structure type definitions are:

```
NewSample_t< int Dimension, int Fred > :=
{
    int[Dimension] Vector ;                               (o)

    Pet_t ObnoxiousPet ;                                   (o)

    Stuff_t<Fred> Thingy ;                                 (o)
} ;

Stuff_t< int George > :=
{
    real[George] IrrelevantStuff ;                         (r)
} ;
```

`NewSample_t` and `Stuff_t` are parameterized structure types. `Dimension` and `Fred` are the structure parameters of `NewSample_t`. `George` is the structure parameter of `Stuff_t`. All structure parameters in this example are of type `int`. `Thingy` is a structure entity of type `Stuff_t`; it uses the parameter `Fred` to complete its declaration. Note the use of `George` and `Fred` in the above example. `George` is a parameter in the definition of `Stuff_t`; `Fred` is an argument in the declaration of an entity of type `Stuff_t`. This mimics the use of parameters in function definitions in C.

A second language feature required to cope with the cascade of information within the hierarchy is structure-related functions. For example, the size of an array within a given structure may be a function of one or more of the structure-defining parameters, or the array size may be a function of

an optional field within the structure. No new syntax is provided to incorporate structure-related functions; they are instead described in terms of their return values, dependencies, and functionality.

An additional notation used in structure typedefs is that each element or field within a structure definition is identified as required, optional, or optional with a default if absent; these are designated by '(r)', '(o)', and '(o/d)', respectively, in the right margin of the structure definition. These designations are included to assist in implementation of the data structures into an actual database and can be used to guide mapping of data as well as error checking. 'Required' fields are those essential to the interpretation of the information contained within the data structure. 'Optional' fields are those that are not necessary but potentially useful, such as documentation. 'Defaulted-optional' fields are those that take on a known default if absent from the database.

In the example of `Sample_t` above, only the element `Dimension` is required. Both elements `Vector` and `ObnoxiousPet` are optional. This means that in any specific instance of the structure, only `Dimension` must be present. An alternative instance of the entity `Sample1` shown above is the following:

```
Sample_t Sample1 =
  {{
    Dimension = 4 ;
  }} ;
```

None of the entities and types defined in the above examples are actually used in the definition of the SIDS.

As a final note, the reader should be aware that the SIDS is a conceptual description of the form of the data. The actual location of data in the file is determined by the file mapping, defined by the appropriate *File Mapping Manual*.

3.2 Structured Grid Notation and Indexing Conventions

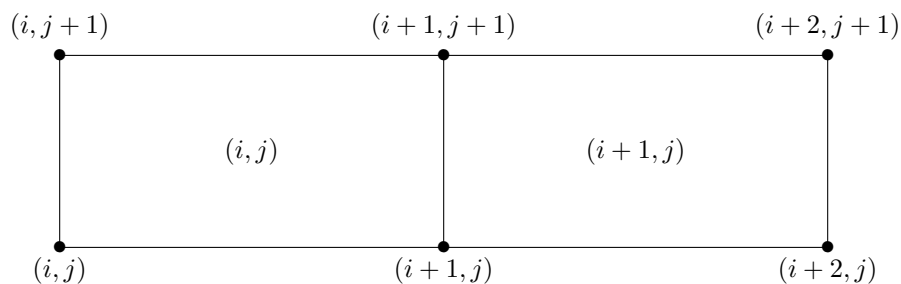
A *grid* is defined by its vertices. In a 3-D structured grid, the volume is the ensemble of cells, where each cell is the hexahedron region defined by eight nearest neighbor vertices. Each cell is bounded by six faces, where each face is the quadrilateral made up of four vertices. An edge links two nearest-neighbor vertices; a face is bounded by four edges.

In a 2-D structured grid, the notation is more ambiguous. Typically, the quadrilateral area composed of four nearest-neighbor vertices is referred to as a cell. The sides of each cell, the line linking two vertices, is either a face or an edge. In a 1-D grid, the line connecting two vertices is a cell.

A *structured multizone grid* is composed of multiple regions called *zones*, where each zone includes all the vertices, cells, faces, and edges that constitute the grid in that region.

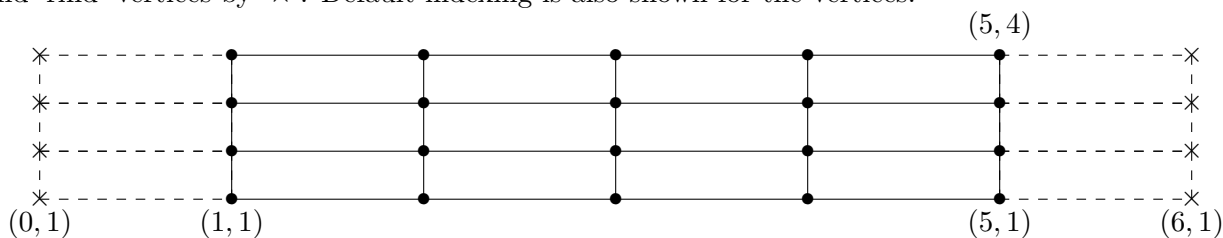
Indices describing a 3-D grid are ordered (i, j, k) ; (i, j) is used for 2-D and (i) for 1-D.

Cell centers, face centers, and edge centers are indexed by the minimum i , j , and k indices of the connecting vertices. For example, a 2-D cell center (or face center on a 3-D grid) would have the following convention:



In addition, the default beginning vertex for the grid in a given zone is $(1, 1, 1)$; this means the default beginning cell center of the grid in that zone is also $(1, 1, 1)$.

A zone may contain grid-coordinate or flow-solution data defined at a set of points outside the zone itself. These are referred to as ‘rind’ or ghost points and may be associated with fictitious vertices or cell centers. They are distinguished from the vertices and cells making up the grid within the zone (including its boundary vertices), which are referred to as ‘core’ points. The following is a 2-D zone with a single row of ‘rind’ vertices at the minimum and maximum i -faces. The grid size (i.e. the number of ‘core’ vertices in each direction) is 5×4 . ‘Core’ vertices are designated by ‘•’, and ‘rind’ vertices by ‘×’. Default indexing is also shown for the vertices.



For a zone, the minimum faces in each coordinate direction are denoted i -min, j -min and k -min; the maximum faces are denoted i -max, j -max and k -max. These are the minimum and maximum ‘core’ faces. For example, i -min is the face or grid plane whose core vertices have minimum i index (which if using default indexing is 1).

3.3 Unstructured Grid Element Numbering Conventions

The major difference in the way structured and unstructured grids are recorded is the element definition. In a structured grid, the elements can always be recomputed easily using the computational coordinates, and therefore they are usually not written in the data file. For an unstructured grid, the element connectivity cannot be easily built, so this additional information is generally added to the data file. The element information typically includes the element type or shape, and the list of nodes for each element.

In an unstructured zone, the nodes are ordered from 1 to N , where N is the number of nodes in the zone. An element is defined as a group of one or more nodes, where each node is represented by its index. The elements are indexed from 1 to M within a zone, where M is the total number of elements defined for the zone.

CGNS supports eight element shapes — points, lines, triangles, quadrangles, tetrahedra, pentahedra, pyramids, and hexahedra. Elements describing a volume are referred to as 3-D elements.

Those defining a surface are 2-D elements. Line and point elements are called 1-D and 0-D elements, respectively.

In a 3-D unstructured mesh, the cells are defined using 3-D elements, while the boundary patches may be described using 2-D elements. The complete element definition may include more than just the cells.

Each element shape may have a different number of nodes, depending on whether linear or quadratic interpolation is used. Therefore the name of each type of element is composed of two parts; the first part identifies the element shape, and the second part the number of nodes. Table 1 summarizes the element types supported in CGNS.

Table 1: Element Types in CGNS

Dimensionality of the Element	Shape	Linear Interpolation	Quadratic Interpolation
0-D	Point	NODE	NODE
1-D	Line	BAR_2	BAR_3
2-D	Triangle	TRI_3	TRI_6
	Quadrangle	QUAD_4	QUAD_8, QUAD_9
3-D	Tetrahedron	TETRA_4	TETRA_10
	Pyramid	PYRA_5	PYRA_14
	Pentahedron	PENTA_6	PENTA_15, PENTA_18
	Hexahedron	HEXA_8	HEXA_20, HEXA_27

Any element type not supported by CGNS can be recorded using the CGNS generic element type NGON_n. See Section 7.3 for more detail.

The ordering of the nodes within an element is important. Since the nodes in each element type could be ordered in multiple ways, it is necessary to define numbering conventions. The following sections describe the element numbering conventions used in CGNS.

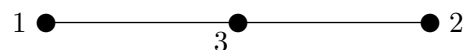
3.3.1 1-D (Line) Elements

1-D elements represent geometrically a line (or bar). The linear form, BAR_2, is composed of two nodes at each extremity of the line. The quadratic form, BAR_3, has an additional node located at the middle of the line.

BAR_2



BAR_3



Face Definition

<u>Oriented edge</u>	<u>Corner nodes</u>	<u>Mid-node</u>
E1	N1, N2	N3

3.3.2 2-D (Surface) Elements

2-D elements represent a surface in either 2-D or 3-D space. Note that in physical space, the surface need not be planar, but may be curved. In a 2-D mesh the elements represent the cells themselves; in a 3-D mesh they represent faces. CGNS supports two shapes of 2-D elements — triangles and quadrangles.

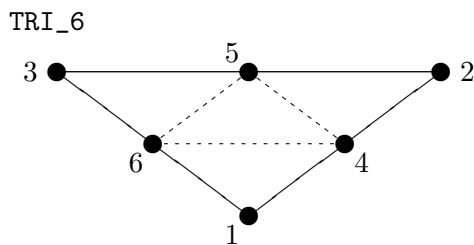
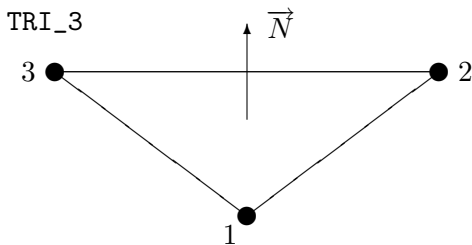
The normal vector of a 2-D element is computed using the cross product of a vector from the first to second node, with a vector from the first to third node. The direction of the normal is such that the three vectors (i.e., $(\vec{N2} - \vec{N1})$, $(\vec{N3} - \vec{N1})$, and $\vec{N}$) form a right-handed triad.

$$\vec{N} = (\vec{N2} - \vec{N1}) \times (\vec{N3} - \vec{N1})$$

In a 2-D mesh, all elements must be oriented the same way; i.e., all normals must point toward the same side of the mesh.

3.3.2.1 Triangular Elements

Two types of triangular elements are supported in CGNS, TRI_3 and TRI_6. TRI_3 elements are composed of three nodes located at the three geometric corners of the triangle. TRI_6 elements have three additional nodes located at the middles of the three edges.



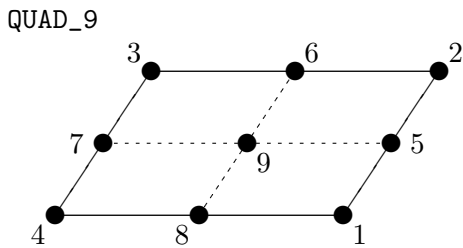
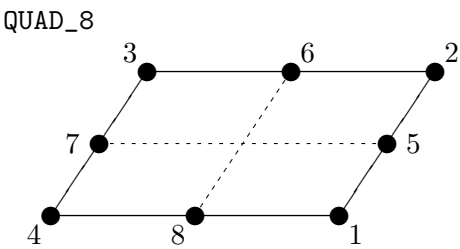
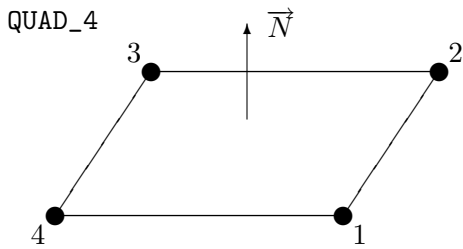
<u>Edge Definition</u>			<u>Face Definition</u>			
<u>Oriented edges</u>	<u>Corner nodes</u>	<u>Mid-node</u>	<u>Face</u>	<u>Corner nodes</u>	<u>Mid-edge nodes</u>	<u>Oriented edges</u>
E1	N1,N2	N4	F1	N1,N2,N3	N4,N5,N6	E1,E2,E3
E2	N2,N3	N5				
E3	N3,N1	N6				

Notes

- N1,...,N6 Grid point identification number. Integer ≥ 0 or blank, and $N1 \neq N2 \neq \dots \neq N6$. Grid points N1, N2, and N3 are in consecutive order about the triangle.
- E1,E2,E3 Edge identification number.
- F1 Face identification number.

3.3.2.2 Quadrilateral Elements

CGNS supports three types of quadrilateral elements, QUAD_4, QUAD_8, and QUAD_9. QUAD_4 elements are composed of four nodes located at the four geometric corners of the quadrangle. In addition, QUAD_8 and QUAD_9 elements have four mid-edge nodes, and QUAD_9 adds a mid-face node.



Edge Definition

<u>Oriented edges</u>	<u>Corner nodes</u>	<u>Mid-node</u>
E1	N1,N2	N5
E2	N2,N3	N6
E3	N3,N4	N7
E4	N4,N1	N8

Face Definition

<u>Face</u>	<u>Corner nodes</u>	<u>Mid-edge nodes</u>	<u>Mid-face node</u>	<u>Oriented edges</u>
F1	N1,N2,N3,N4	N5,N6,N7,N8	N9	E1,E2,E3,E4

Notes

- N1,...,N9 Grid point identification number. Integer ≥ 0 or blank, and $N1 \neq N2 \neq \dots \neq N9$. Grid points N1...N4 are in consecutive order about the quadrangle.
- E1,...,E4 Edge identification number.
- F1 Face identification number.

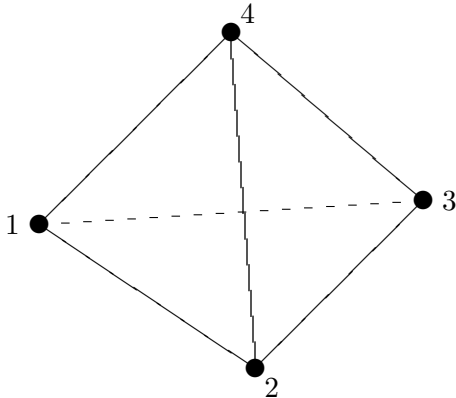
3.3.3 3-D (Volume) Elements

3-D elements represent a volume in 3-D space, and constitute the cells of a 3-D mesh. CGNS supports four different shapes of 3-D elements — tetrahedra, pyramids, pentahedra, and hexahedra.

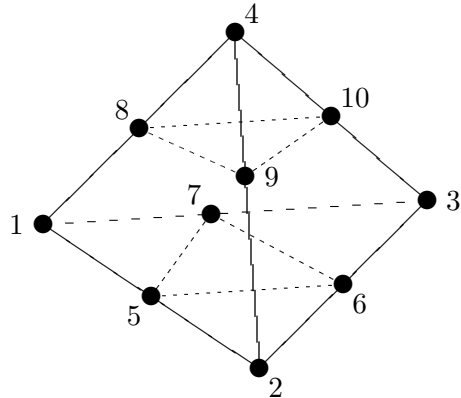
3.3.3.1 Tetrahedral Elements

CGNS supports two types of tetrahedral elements, TETRA_4 and TETRA_10. TETRA_4 elements are composed of four nodes located at the four geometric corners of the tetrahedron. TETRA_10 elements have six additional nodes, at the middle of each of the six edges.

TETRA_4



TETRA_10



Edge Definition

<u>Oriented edges</u>	<u>Corner nodes</u>	<u>Mid-node</u>
E1	N1,N2	N5
E2	N2,N3	N6
E3	N3,N1	N7
E4	N1,N4	N8
E5	N2,N4	N9
E6	N3,N4	N10

Face Definition

<u>Face</u>	<u>Corner nodes</u>	<u>Mid-edge nodes</u>	<u>Oriented edges</u>
F1	N1,N3,N2	N7,N6,N5	-E3,-E2,-E1
F2	N1,N2,N4	N5,N9,N8	E1,E5,-E4
F3	N2,N3,N4	N6,N10,N9	E2,E6,-E5
F4	N3,N1,N4	N7,N8,N10	E3,E4,-E6

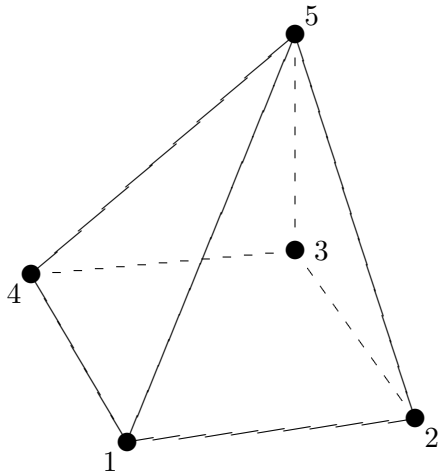
Notes

- N1,...,N10 Grid point identification number. Integer ≥ 0 or blank, and $N1 \neq N2 \neq \dots \neq N10$. Grid points N1...N3 are in consecutive order about one trilateral face. The cross product of a vector going from N1 to N2, with a vector going from N1 to N3, must result in a vector oriented from face F1 toward N4.
- E1,...,E6 Edge identification number. The edges are oriented from the first to the second node. A negative edge (e.g., -E1) means that the edge is used in its reverse direction.
- F1,...,F4 Face identification number. The faces are oriented so that the cross product of a vector from its first to second node, with a vector from its first to third node, is oriented outward.

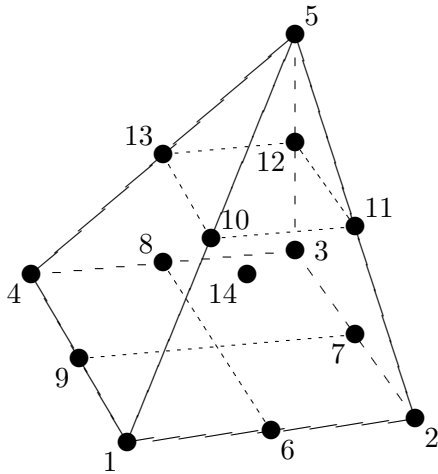
3.3.3.2 Pyramid Elements

CGNS supports two types of pyramid elements, PYRA_5 and PYRA_14. PYRA_5 elements are composed of five nodes located at the five geometric corners of the pyramid. PYRA_14 elements have nine additional nodes, eight located at the middle of each of the eight edges, and one at the cell center.

PYRA_5



PYRA_14



Edge Definition

Oriented edges	Corner nodes	Mid-node
E1	N1,N2	N6
E2	N2,N3	N7
E3	N3,N4	N8
E4	N4,N1	N9
E5	N1,N5	N10
E6	N2,N5	N11
E7	N3,N5	N12
E8	N4,N5	N13

Face Definition

Face	Corner nodes	Mid-edge nodes	Oriented edges
F1	N1,N4,N3,N2	N9,N8,N7,N6	-E4,-E3,-E2,-E1
F2	N1,N2,N5	N6,N11,N10	E1,E6,-E5
F3	N2,N3,N5	N7,N12,N11	E2,E7,-E6
F4	N3,N4,N5	N8,N13,N12	E3,E8,-E7
F5	N4,N1,N5	N9,N10,N13	E4,E5,-E8

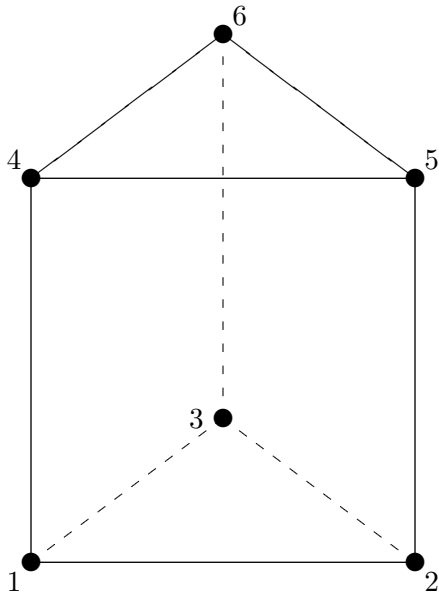
Notes

- N1,...,N14 Grid point identification number. Integer ≥ 0 or blank, and $N1 \neq N2 \neq \dots \neq N14$. Grid points N1...N4 are in consecutive order about the quadrilateral face. The cross product of a vector going from N1 to N2, with a vector going from N1 to N3, must result in a vector oriented from face F1 toward N5. N14 is located at the cell center.
- E1,...,E8 Edge identification number. The edges are oriented from the first to the second node. A negative edge (e.g., -E1) means that the edge is used in its reverse direction.
- F1,...,F5 Face identification number. The faces are oriented so that the cross product of a vector from its first to second node, with a vector from its first to third node, is oriented outward.

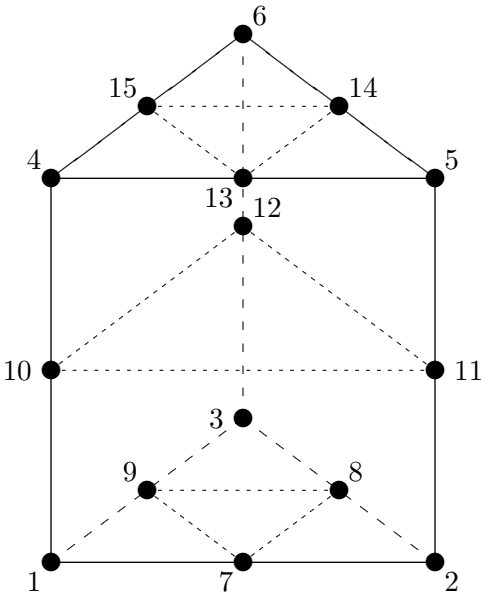
3.3.3.3 Pentahedral Elements

CGNS supports three types of pentahedral elements, PENTA_6, PENTA_15, and PENTA_18. PENTA_6 elements are composed of six nodes located at the six geometric corners of the pentahedron. In addition, PENTA_15 and PENTA_18 elements have a node at the middle of each of the nine edges; PENTA_18 adds a node at the middle of each of the three quadrilateral faces.

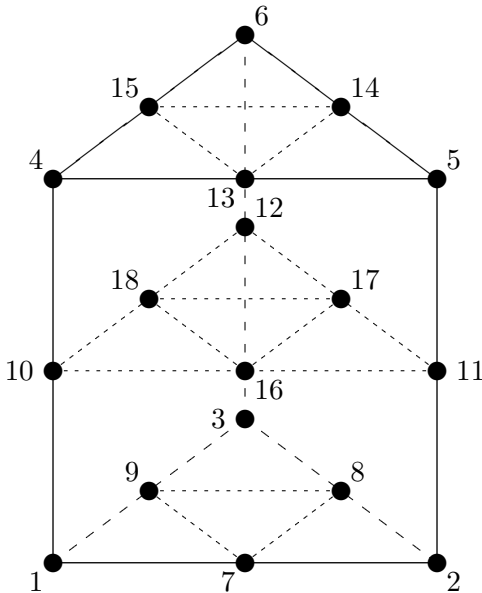
PENTA_6



PENTA_15



PENTA_18



Edge Definition

<u>Oriented edges</u>	<u>Corner nodes</u>	<u>Mid-node</u>
E1	N1,N2	N7
E2	N2,N3	N8
E3	N3,N1	N9
E4	N1,N4	N10
E5	N2,N5	N11
E6	N3,N6	N12
E7	N4,N5	N13
E8	N5,N6	N14
E9	N6,N4	N15

Face Definition

Face	Corner nodes	Mid-edge nodes	Mid-face node	Oriented edges
F1	N1,N2,N5,N4	N7, N11,N13,N10	N16	E1, E5,-E7,-E4
F2	N2,N3,N6,N5	N8, N12,N14,N11	N17	E2, E6,-E8,-E5
F3	N3,N1,N4,N6	N9, N10,N15,N12	N18	E3, E4,-E9,-E6
F4	N1,N3,N2	N9, N8, N7		-E3,-E2,-E1
F5	N4,N5,N6	N13,N14,N15		E7, E8, E9

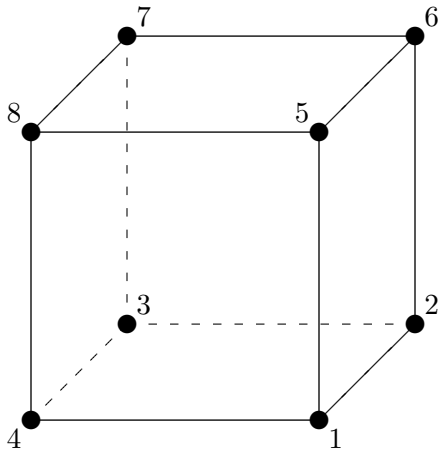
Notes

- N1,...,N18 Grid point identification number. Integer ≥ 0 or blank, and $N1 \neq N2 \neq \dots \neq N18$. Grid points N1...N3 are in consecutive order about one trilateral face. Grid points N4...N6 are in order in the same direction around the opposite trilateral face.
- E1,...,E9 Edge identification number. The edges are oriented from the first to the second node. A negative edge (e.g., -E1) means that the edge is used in its reverse direction.
- F1,...,F5 Face identification number. The faces are oriented so that the cross product of a vector from its first to second node, with a vector from its first to third node, is oriented outward.

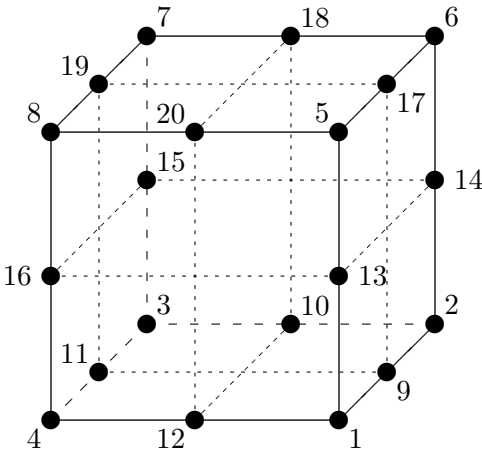
3.3.3.4 Hexahedral Elements

CGNS supports three types of hexahedral elements, HEXA_8, HEXA_20, and HEXA_27. HEXA_8 elements are composed of eight nodes located at the eight geometric corners of the hexahedron. In addition, HEXA_20 and HEXA_27 elements have a node at the middle of each of the twelve edges; HEXA_27 adds a node at the middle of each of the six faces, and one at the cell center.

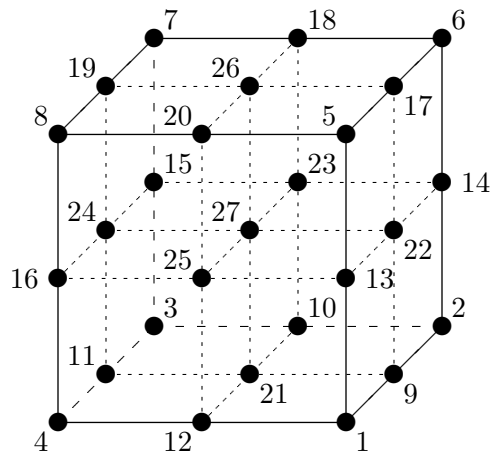
HEXA_8



HEXA_20



HEXA_27

Edge Definition

<u>Oriented edges</u>	<u>Corner nodes</u>	<u>Mid-node</u>
E1	N1,N2	N9
E2	N2,N3	N10
E3	N3,N4	N11
E4	N4,N1	N12
E5	N1,N5	N13
E6	N2,N6	N14
E7	N3,N7	N15
E8	N4,N8	N16
E9	N5,N6	N17
E10	N6,N7	N18
E11	N7,N8	N19
E12	N8,N5	N20

Face Definition

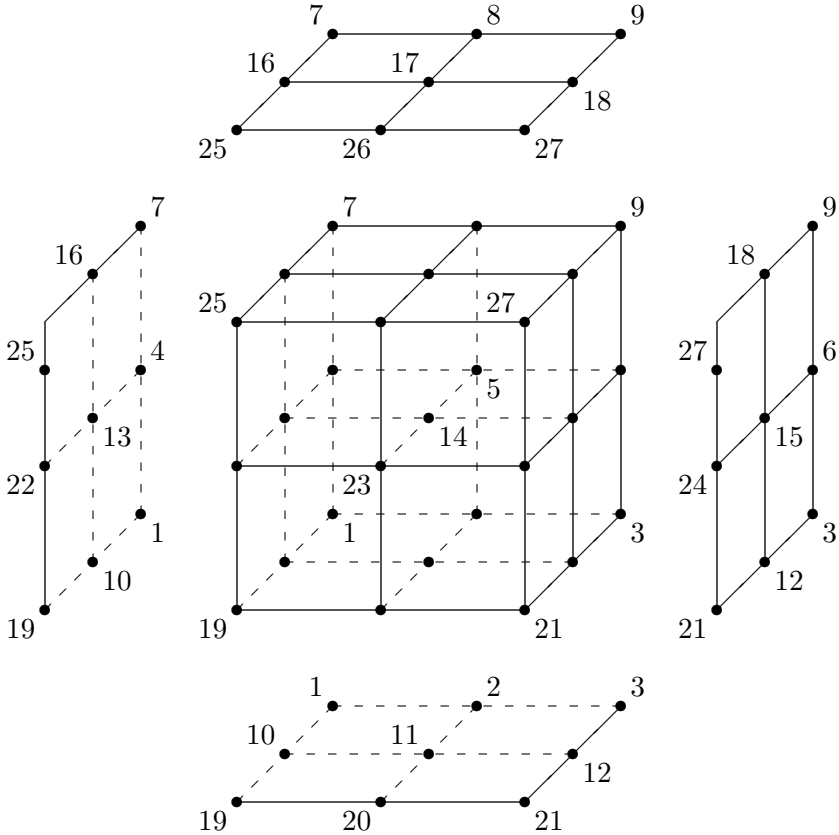
<u>Face</u>	<u>Corner nodes</u>	<u>Mid-edge nodes</u>	<u>Mid-face node</u>	<u>Oriented edges</u>
F1	N1,N4,N3,N2	N12,N11,N10,N9	N21	-E4,-E3, -E2, -E1
F2	N1,N2,N6,N5	N9, N14,N17,N13	N22	E1, E6, -E9, -E5
F3	N2,N3,N7,N6	N10,N15,N18,N14	N23	E2, E7, -E10,-E6
F4	N3,N4,N8,N7	N11,N16,N19,N15	N24	E3, E8, -E11,-E7
F5	N1,N5,N8,N4	N13,N20,N16,N12	N25	E5,-E12,-E8, E4
F6	N5,N6,N7,N8	N17,N18,N19,N20	N26	E9, E10, E11, E12

Notes

- N1, ..., N27 Grid point identification number. Integer ≥ 0 or blank, and $N1 \neq N2 \neq \dots \neq N27$. Grid points N1...N4 are in consecutive order about one quadrilateral face. Grid points N5...N8 are in order in the same direction around the opposite quadrilateral face.
- E1, ..., E12 Edge identification number. The edges are oriented from the first to the second node. A negative edge (e.g., -E1) means that the edge is used in its reverse direction.
- F1, ..., F6 Face identification number. The faces are oriented so that the cross product of a vector from its first to second node, with a vector from its first to third node, is oriented outward.

3.3.4 Unstructured Grid Example

Consider an unstructured zone in the shape of a cube, with each edge of the zone having three nodes. The resulting unstructured grid has a total of 27 nodes, as illustrated in the exploded figure below.



This zone contains eight hexahedral cells, numbered 1 to 8, and the cell connectivity is:

Element No.	Element Connectivity
1	1, 2, 5, 4, 10, 11, 14, 13
2	2, 3, 6, 5, 11, 12, 15, 14
3	4, 5, 8, 7, 13, 14, 17, 16
4	5, 6, 9, 8, 14, 15, 18, 17
5	10, 11, 14, 13, 19, 20, 23, 22
6	11, 12, 15, 14, 20, 21, 24, 23
7	13, 14, 17, 16, 22, 23, 26, 25
8	14, 15, 18, 17, 23, 24, 27, 26

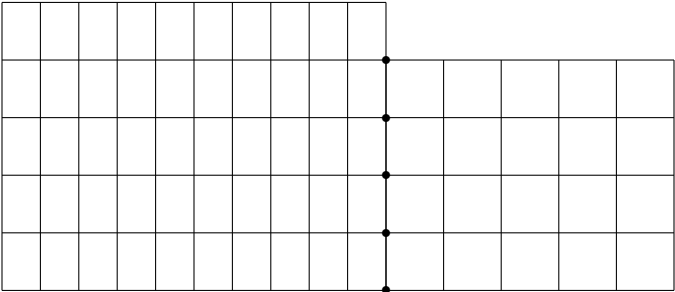
In addition to the cells, the boundary faces could also be added to the element definition of this unstructured zone. There are 24 boundary faces in this zone, corresponding to element numbers 9 to 32. Each boundary face is of type **QUAD_4**. The table below shows the element connectivity of each boundary face, as well as the element number and face number of its parent cell.

Face	Element No.	Element Connectivity	Parent Cell	Parent Face
Left	9	1, 10, 13, 4	1	5
	10	4, 13, 16, 7	3	5
	11	10, 19, 22, 13	5	5
	12	13, 22, 25, 16	7	5
Right	13	3, 6, 15, 12	2	3
	14	6, 9, 18, 15	4	3
	15	12, 15, 24, 21	6	3
	16	15, 18, 27, 24	8	3
Bottom	17	1, 2, 11, 10	1	2
	18	2, 3, 12, 11	2	2
	19	10, 11, 20, 19	5	2
	20	11, 12, 21, 20	6	2
Top	21	7, 16, 17, 8	3	4
	22	8, 17, 18, 9	4	4
	23	16, 25, 26, 17	7	4
	24	17, 26, 27, 18	8	4
Back	25	1, 4, 5, 2	1	1
	26	2, 5, 6, 3	2	1
	27	4, 7, 8, 5	3	1
	28	5, 8, 9, 6	4	1
Front	29	19, 20, 23, 22	5	6
	30	20, 21, 24, 23	6	6
	31	22, 23, 26, 25	7	6
	32	23, 24, 27, 26	8	6

3.4 Multizone Interfaces

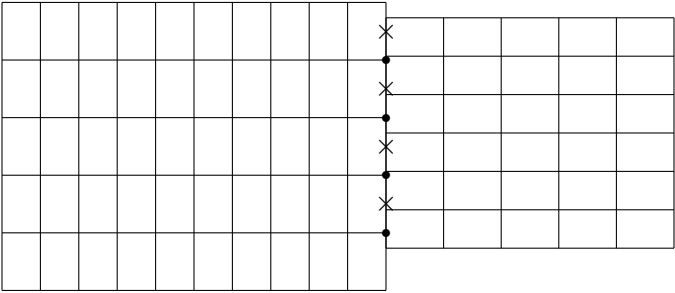
Figure 2 depicts three types of multizone interfaces, shown for structured zones. The first type is a 1-to-1 abutting interface, also referred to as matching or C0 continuous. The interface is a plane of vertices that are physically coincident between the adjacent zones. For structured zones, grid-coordinate lines perpendicular to the interface are continuous from one zone to the next; in 3-D, a 1-to-1 abutting interface is usually a logically rectangular region.

The second type of interface is mismatched abutting, where two zones touch but do not overlap (except for vertices and cell faces on the grid plane of the interface). Vertices on the interface may not be physically coincident between the two zones. Figure 2b identifies the vertices and face



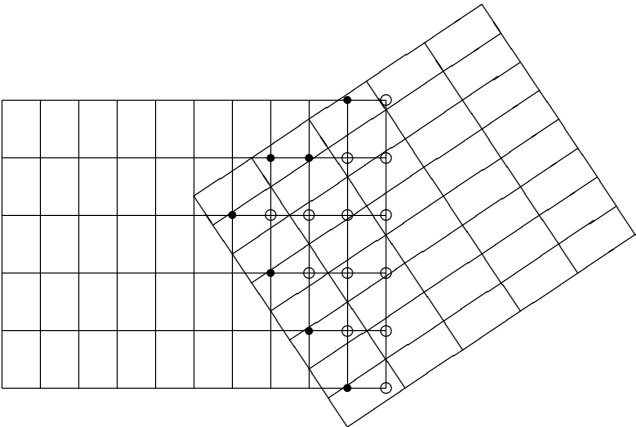
- Left-zone vertices on interface

(a) 1-to-1 Abutting Interface



- Left-zone vertices on interface
- × Left-zone face-centers on interface

(b) Mismatched Abutting Interface



- Left-zone fringe points (vertices)
- Left-zone overset-hole points (vertices)

(c) Overset Interface

Figure 2: Structured-Grid Multizone Interface Types

centers of the left zone that lay on the interface. Even for structured zones in 3-D, the vertices of a zone that constitute an interface patch may not form a logically rectangular region.

The third type of multizone interface is called overset and occurs when two zones overlap; in 3-D, the overlap is a 3-D region. For overset interfaces, one of the two zones takes precedence over the other; this establishes which solution in the overlap region to retain and which to discard. The region in a given zone where the solution is discarded is called an overset hole and the grid points outlining the hole are called fringe points. Figure 2c depicts an overlap region between two zones. The right zone takes precedence over the left zone, and the points identified in the figure are the fringe points and overset-hole points for the left zone. In addition, for the zone taking precedence, any bounding points (i.e. vertices on the bounding faces) of the zone that lay within the overlap region must also be identified.

Overset interfaces may also include multiple layers of fringe points outlining holes and at zone boundaries.

For the mismatched abutting and overset interfaces in Figure 2, the left zone plays the role of receiver zone and the right zone plays the role of donor zone.

4 Building-Block Structure Definitions

This section defines and describes low-level structures types that are used in the definition of more complex structures within the hierarchy.

4.1 Definition: DataClass_t

DataClass_t is an enumeration type that identifies the class of a given piece of data.

```
DataClass_t := Enumeration(
    Null,
    Dimensional,
    NormalizedByDimensional,
    NormalizedByUnknownDimensional,
    NondimensionalParameter,
    DimensionlessConstant,
    UserDefined ) ;
```

These classes divide data into different categories depending on dimensional units or normalization associated with the data. **Dimensional** specifies dimensional data. **NormalizedByDimensional** specifies nondimensional data that is normalized by dimensional reference quantities. In contrast, **NormalizedByUnknownDimensional** specifies nondimensional data typically found in completely nondimensional databases, where all field and reference data is nondimensional. **NondimensionalParameter** specifies nondimensional parameters such as Mach number and lift coefficient. Constants such as π are designated by **DimensionlessConstant**. The distinction between these different classes is further discussed in Section 5.

4.2 Definition: Descriptor_t

Descriptor_t is a documentation or annotation structure which contains a character string. Characters allowed within the string include newlines, tabs and other special characters; this potentially allows for unlimited documentation inclusion within the database. For example, a single **Descriptor_t** structure could be used to ‘swallow’ an entire ASCII file. In the hierarchical structures defined in the next sections, each allows for the inclusion of multiple **Descriptor_t** substructures. Conventions could be made for names of often-used **Descriptor_t** structure entities, such as **ReadMe** or **YouReallyWantToReadMeFirst**.

```
Descriptor_t :=
{
    Data(char, 1, string_length) ;           (r)
} ;
```

where **string_length** is the length of the character string.

4.3 Definition: DimensionalUnits_t

DimensionalUnits_t describes the system of units used to measure dimensional data. It is composed of a set of enumeration types that define the units for mass, length, time, temperature, angle, electric current, substance amount, and luminous intensity.

```

MassUnits_t           := Enumeration( Null, Kilogram, Gram, Slug, PoundMass,
                                      UserDefined ) ;

LengthUnits_t         := Enumeration( Null, Meter, Centimeter, Millimeter,
                                      Foot, Inch, UserDefined ) ;

TimeUnits_t           := Enumeration( Null, Second, UserDefined ) ;

TemperatureUnits_t    := Enumeration( Null, Kelvin, Celsius, Rankine,
                                      Fahrenheit, UserDefined ) ;

AngleUnits_t          := Enumeration( Null, Degree, Radian, UserDefined ) ;

ElectricCurrentUnits_t := Enumeration( Null, Ampere, Abampere, Statampere,
                                      Edison, auCurrent, UserDefined ) ;

SubstanceAmountUnits_t := Enumeration( Null, Mole, Entities,
                                      StandardCubicFoot,
                                      StandardCubicMeter, UserDefined ) ;

LuminousIntensityUnits_t := Enumeration( Null, Candela, Candle, Carcel,
                                      Hefner, Violle, UserDefined ) ;

DimensionalUnits_t :=
{
  MassUnits_t      MassUnits ;           (r)
  LengthUnits_t    LengthUnits ;         (r)
  TimeUnits_t      TimeUnits ;           (r)
  TemperatureUnits_t TemperatureUnits ;  (r)
  AngleUnits_t     AngleUnits ;          (r)
  AdditionalUnits_t :=                   (o)
  {
    ElectricCurrentUnits_t ElectricCurrentUnits ; (r)
    SubstanceAmountUnits_t SubstanceAmountUnits ; (r)
    LuminousIntensityUnits_t LuminousIntensityUnits ; (r)
  }
} ;

```

The International System (SI) uses the following units.

Physical Quantity	Unit
Mass	Kilogram
Length	Meter
Time	Second
Temperature	Kelvin
Angle	Radian
Electric Current	Ampere
Substance Amount	Mole
Luminous Intensity	Candela

For an entity of type `DimensionalUnits_t`, if all the elements of that entity have the value `Null` (i.e., `MassUnits = Null`, etc.), this is equivalent to stating that the data described by the entity is nondimensional.

4.4 Definition: `DimensionalExponents_t`

`DimensionalExponents_t` describes the dimensionality of data by defining the exponents associated with each of the fundamental units.

```

DimensionalExponents_t :=
{
  real MassExponent ;                               (r)
  real LengthExponent ;                             (r)
  real TimeExponent ;                               (r)
  real TemperatureExponent ;                         (r)
  real AngleExponent ;                              (r)
  AdditionalExponents_t :=                          (o)
  {
    real ElectricCurrentExponent ;                   (r)
    real SubstanceAmountExponent ;                   (r)
    real LuminousIntensityExponent ;                  (r)
  }
} ;

```

For example, an instance of `DimensionalExponents_t` that describes velocity is,

```

DimensionalExponents_t =
{{
  MassExponent      = 0 ;
  LengthExponent    = +1 ;
  TimeExponent      = -1 ;
  TemperatureExponent = 0 ;
  AngleExponent     = 0 ;
}} ;

```

4.5 Definition: GridLocation_t

GridLocation_t identifies locations with respect to the grid; it is an enumeration type.

```
GridLocation_t := Enumeration(
    Null,
    Vertex,
    CellCenter,
    FaceCenter,
    IFaceCenter,
    JFaceCenter,
    KFaceCenter,
    EdgeCenter,
    UserDefined ) ;
```

Vertex is coincident with the grid vertices. CellCenter is the center of a cell; this is also appropriate for entities associated with cells but not necessarily with a given location in a cell. For structured zones, IFaceCenter is the center of a face in 3-D whose computational normal points in the i direction. JFaceCenter and KFaceCenter are similarly defined, again only for structured zones. FaceCenter is the center of a generic face which can point in any coordinate direction. These are also appropriate for entities associated with a face, but not located at a specific place on the face. EdgeCenter is the center of an edge. See Section 3.2 for descriptions of cells, faces and edges.

All of the entities of type GridLocation_t defined in this document use a default value of Vertex.

4.6 Definition: IndexArray_t

IndexArray_t specifies an array of indices. An argument is included that allows for specifying the data type of each index; typically the data type will be integer (int). IndexArray_t defines an array of indices of size ArraySize, where the dimension of each index is IndexDimension.

```
IndexArray_t< int IndexDimension, int ArraySize, DataType > :=
{
    Data( DataType, 2, [IndexDimension,ArraySize] ) ;           (r)
} ;
```

4.7 Definition: IndexRange_t

IndexRange_t specifies the beginning and ending indices of a subrange. The subrange may describe a portion of a grid line, grid plane, or grid volume.

```
IndexRange_t< int IndexDimension > :=
{
    int[IndexDimension] Begin ;                                   (r)
    int[IndexDimension] End ;                                     (r)
} ;
```

4.8 Definition: Rind_t

```

Rind_t< int IndexDimension > :=
{
  int[2*IndexDimension] RindPlanes ;
} ;

```

(r)

$n = 1 \rightarrow i\text{-min}$	$n = 2 \rightarrow i\text{-max}$
$n = 3 \rightarrow j\text{-min}$	$n = 4 \rightarrow j\text{-max}$
$n = 5 \rightarrow k\text{-min}$	$n = 6 \rightarrow k\text{-max}$

$$\begin{aligned} i: & (1 - a, II + b) \\ j: & (1 - c, JJ + d) \\ k: & (1 - e, KK + f) \end{aligned}$$

5 Data-Array Structure Definitions

This section defines the structure type `DataArray_t` for describing data arrays. This general-purpose structure is used to declare data arrays and scalars throughout the CGNS hierarchy. It is used to describe grid coordinates, flow-solution data, governing flow parameters, boundary-condition data, and other information. For most of these different types of CFD data, we have also established a list of standardized identifiers for entities of type `DataArray_t`. For example, `Density` is used for data arrays containing static density. The list of standardized data-name identifiers is provided in Annex A.

We address five classes of data with the `DataArray_t` structure type:

- (a) dimensional data (e.g. velocity in units of m/s);
- (b) nondimensional data normalized by dimensional reference quantities;
- (c) nondimensional data with associated nondimensional reference quantities;
- (d) nondimensional parameters (e.g. Reynolds number, pressure coefficient);
- (e) pure constants (e.g. π , e).

The first two of these classes often occur within the same test case, where each piece of data is either dimensional itself or normalized by a dimensional quantity. The third data class is typical of a completely nondimensional test case, where all field data and reference quantities are nondimensional. The fourth class, nondimensional parameters, are universal in CFD, although not always consistently defined. The individual components of nondimensional parameters may be data from any of the first three classes.

Each of the five classes of data requires different information to describe dimensional units or normalization associated with the data. These requirements are reflected in the structure definition for `DataArray_t`.

The remainder of this section is as follows: the structure type `DataArray_t` is first defined. Then the class identification and data manipulation is discussed in Section 5.2 for each of the five data classes. Finally, examples of `DataArray_t` entities are presented in Section 5.3.

5.1 Definition: `DataArray_t`

`DataArray_t` describes a multi-dimensional data array of given type, dimensionality and size in each dimension. The data may be dimensional, nondimensional or pure constants. Qualifiers are provided to describe dimensional units or normalization information associated with the data.

```
DataArray_t< DataType, int Dimension, int[Dimension] DimensionValues > :=
{
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                               (o)

  Data( DataType, Dimension, DimensionValues ) ;                               (r)
```

```

DataClass_t DataClass ; (o)

DimensionalUnits_t DimensionalUnits ; (o)

DimensionalExponents_t DimensionalExponents ; (o)

DataConversion_t DataConversion ; (o)
} ;

```

Notes

1. Default names for the **Descriptor_t** list are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of **DataArray_t** and shall not include the names **DataClass**, **DimensionalUnits**, **DimensionalExponents**, or **DataConversion**.
2. **Data()** is the only required field for **DataArray_t**.

DataArray_t requires three structure parameters: **Dimension** is the dimensionality of the data array; **DimensionValues** is an array of length **Dimension** that contains the size of the data arrays in each dimension; and **DataType** is the data type of the data stored. **DataType** will usually be **real**, but other data types are permissible.

The optional entities **DataClass**, **DimensionalUnits**, **DimensionalExponents** and **DataConversion** provide information on dimensional units and normalization associated with the data. The function of these qualifiers is provided in the next section.

This structure type is formulated to describe an array of scalars. Therefore, for vector quantities (e.g. the position vector or the velocity vector), separate structure entities are required for each component of the vector. For example, the cartesian coordinates of a 3-D grid are described by three separate **DataArray_t** entities: one for *x*, one for *y* and one for *z* (see Example 7-A).

5.1.1 Definition: DataConversion_t

DataConversion_t contains conversion factors for recovering raw dimensional data from given nondimensional data. These conversion factors are typically associated with nondimensional data that is normalized by dimensional reference quantities.

```

DataConversion_t :=
{
  real ConversionScale ; (r)

  real ConversionOffset ; (r)
} ;

```

Given a nondimensional piece of data, **Data(nondimensional)**, the conversion to 'raw' dimensional form is:

$$\text{Data}(\text{raw}) = \text{Data}(\text{nondimensional}) * \text{ConversionScale} + \text{ConversionOffset}$$

These conversion factors are further described in Section 5.2.2.

5.2 Data Manipulation

The optional entities of **DataArray_t** provide information for manipulating the data, including changing units or normalization. This section describes the rules under which these optional entities operate and the specific manipulations that can be performed on the data.

Within a given instance of **DataArray_t**, the class of data and all information required for manipulations may be completely and precisely specified by the entities **DataClass**, **DimensionalUnits**, **DimensionalExponents** and **DataConversion**. **DataClass** identifies the class of data and governs the manipulations that can be performed. Each of the five data classes is treated separately in the subsequent sections.

The entities **DataClass** and **DimensionalUnits** serve special functions in the CGNS hierarchy. If **DataClass** is absent from a given instance of **DataArray_t**, then its value is determined from ‘global’ data. This global data may be set at any level of the CGNS hierarchy with the data set at the lowest level taking precedence. **DimensionalUnits** may be similarly set by global data. The rules for determining the appropriate set of global data to apply is further detailed in Section 6.4.

This alternate functionality provides a measure of economy in describing dimensional units or normalization within the hierarchy. Examples that make use of global data are presented in Section 7.2 and Section 7.8 for grid and flow solution data. The complete two-zone case of Annex B also depicts this alternate functionality.

5.2.1 Dimensional Data

If **DataClass** = **Dimensional**, the data is dimensional. The optional qualifiers **DimensionalUnits** and **DimensionalExponents** describe dimensional units associated with the data. These qualifiers are provided to specify the system of dimensional units and the dimensional exponents, respectively. For example, if the data is the *x*-component of velocity, then **DimensionalUnits** will state that the pertinent dimensional units are, say, **Meter** and **Second**; **DimensionalExponents** will specify the pertinent dimensional exponents are **LengthExponent** = 1 and **TimeExponent** = -1. Combining the information gives the units m/s. Examples showing the use of these two qualifiers are provided in Section 5.3.

If **DimensionalUnits** is absent, then the appropriate set of dimensional units is obtained from ‘global’ data. The rules for determining this appropriate set of ‘global’ dimensional units are presented in Section 6.4.

If **DimensionalExponents** is absent, then the appropriate dimensional exponents can be determined by convention if the specific instance of **DataArray_t** corresponds to one of the standardized data-name identifiers listed in Annex A. Otherwise, the exponents are unspecified. We strongly recommend inclusion of the **DimensionalExponents** qualifier whenever the data is dimensional and the instance of **DataArray_t** is not among the list of standardized identifiers.

5.2.2 Nondimensional Data Normalized by Dimensional Quantities

If `DataClass = NormalizedByDimensional`, the data is nondimensional and is normalized by dimensional reference quantities. All optional entities in `DataArray_t` are used. `DataConversion` contains factors to convert the nondimensional data to 'raw' dimensional data; these factors are `ConversionScale` and `ConversionOffset`. The conversion process is as follows:

$$\text{Data}(\text{raw}) = \text{Data}(\text{nondimensional}) * \text{ConversionScale} + \text{ConversionOffset}$$

where `Data(nondimensional)` is the original nondimensional data, and `Data(raw)` is the converted raw data. This converted raw data is dimensional, and the optional qualifiers `DimensionalUnits` and `DimensionalExponents` describe the appropriate dimensional units and exponents. Note that `DimensionalUnits` and `DimensionalExponents` also describe the units for `ConversionScale` and `ConversionOffset`.

If `DataConversion` is absent, the equivalent defaults are `ConversionScale = 1` and `ConversionOffset = 0`. If either `DimensionalUnits` or `DimensionalExponents` is absent, follow the rules described in the previous section.

Note that functionally there is little difference between these first two data classes (`DataClass = Dimensional` and `NormalizedByDimensional`). In the first case the data is dimensional, and in the second, the converted raw data is dimensional. Also, the equivalent defaults for `DataConversion` produce no changes in the data; hence, it is almost the same as stating the original data is dimensional.

5.2.3 Nondimensional Data Normalized by Unknown Dimensional Quantities

If `DataClass = NormalizedByUnknownDimensional`, the data is nondimensional and is normalized by some unspecified dimensional quantities. This type of data is typical of a completely nondimensional test case, where all field data and all reference quantities are nondimensional.

Only the `DimensionalExponents` qualifier is used in this case, although it is expected that this qualifier will be seldom utilized in practice. For entities of `DataArray_t` that are not among the list of standardized data-name identifiers, the qualifier could provide useful information by defining the exponents of the dimensional form of the nondimensional data.

Rather than providing qualifiers to describe the normalization of the data, we instead dictate that all data of type `NormalizedByUnknownDimensional` in a given database be nondimensionalized consistently. This is done by picking one set of mass, length, time and temperature scales and normalizing all appropriate data by these scales. We describe this process in detail in the following. Annex B also shows a completely nondimensional database where consistent normalization is used throughout.

The practice of nondimensionalization within flow solvers and other application codes is quite popular. The problem with this practice is that to manipulate the data from a given code, one must often know the particulars of the nondimensionalization used. This largely results from what we call inconsistent normalization—more than the minimum required scales are used to normalize data

within the code. For example, in the OVERFLOW flow solver, the following nondimensionalization is used:

$$\begin{aligned}\tilde{x} &= x/L, & \tilde{u} &= u/c_\infty, & \tilde{\rho} &= \rho/\rho_\infty, \\ \tilde{y} &= y/L, & \tilde{v} &= v/c_\infty, & \tilde{p} &= p/(\rho_\infty c_\infty^2), \\ \tilde{z} &= z/L, & \tilde{w} &= w/c_\infty, & \tilde{\mu} &= \mu/\mu_\infty,\end{aligned}$$

where (x, y, z) are the cartesian coordinates, (u, v, w) are the cartesian components of velocity, ρ is static density, p is static pressure, c is the static speed of sound, and μ is the molecular viscosity. In this example, tilde quantities ($\tilde{\cdot}$) are nondimensional and all others are dimensional. Four dimensional scales are used for normalization: L (a unit length), ρ_∞ , c_∞ and μ_∞ . However, only three fundamental dimensional units are represented: mass, length and time. The extra normalizing scale leads to inconsistent normalization. The primary consequence of this is additional nondimensional parameters, such as Reynolds number, appearing in the nondimensionalized governing equations where none are found in the original dimensional equations. Many definitions, including skin friction coefficient, also have extra terms appearing in the nondimensionalized form. This adds unnecessary complication to any data or equation manipulation associated with the flow solver.

Consistent normalization avoids many of these problems. Here the number of scales used for normalization is the same as the number of fundamental dimensional units represented by the data. Using consistent normalization, the resulting nondimensionalized form of equations and definitions is identical to their original dimensional formulations. One piece of evidence to support this assertion is that it is not possible to form any nondimensional parameters from the set of dimensional scales used for normalization.

An important fallout of consistent normalization is that the actual scales used for normalization become immaterial for all data manipulation processes. To illustrate this consider the following nondimensionalization procedure: let M (mass), L (length) and T (time) be arbitrary dimensional scales by which all data is normalized (neglect temperature data for the present). The nondimensional data follows:

$$\begin{aligned}x' &= x/L, & u' &= u/(L/T), & \rho' &= \rho/(M/L^3), \\ y' &= y/L, & v' &= v/(L/T), & p' &= p/(M/(LT^2)), \\ z' &= z/L, & w' &= w/(L/T), & \mu' &= \mu/(M/(LT)),\end{aligned}$$

where primed quantities are nondimensional and all others are dimensional.

Consider an existing database where all field data and all reference data is nondimensional and normalized as shown. Assume the database has a single reference state given by,

$$\begin{aligned}x'_{\text{ref}} &= x_{\text{ref}}/L, & u'_{\text{ref}} &= u_{\text{ref}}/(L/T), & \rho'_{\text{ref}} &= \rho_{\text{ref}}/(M/L^3), \\ y'_{\text{ref}} &= y_{\text{ref}}/L, & v'_{\text{ref}} &= v_{\text{ref}}/(L/T), & p'_{\text{ref}} &= p_{\text{ref}}/(M/(LT^2)), \\ z'_{\text{ref}} &= z_{\text{ref}}/L, & w'_{\text{ref}} &= w_{\text{ref}}/(L/T), & \mu'_{\text{ref}} &= \mu_{\text{ref}}/(M/(LT)).\end{aligned}$$

If a user wanted to change the nondimensionalization of grid-point pressures, the procedure is straightforward. Let the desired new normalization be given by $p''_{ijk} = p_{ijk}/(\rho_{\text{ref}} c_{\text{ref}}^2)$, where all terms on the right-hand-side are *dimensional*, and as such they are unknown to the database user. However, the desired manipulation is possible using only nondimensional data provided in the

database,

$$\begin{aligned}
 p_{ijk}'' &\equiv p_{ijk}/(\rho_{\text{ref}} c_{\text{ref}}^2) \\
 &= \frac{p_{ijk}}{M/(LT^2)} \frac{M/L^3}{\rho_{\text{ref}}} \left[\frac{L/T}{c_{\text{ref}}} \right]^2 \\
 &= p_{ijk}'/(\rho_{\text{ref}}' (c_{\text{ref}}')^2)
 \end{aligned}$$

Thus, the desired renormalization is possible using the database's nondimensional data as if it were actually dimensional. There is, in fact, a high degree of equivalence between dimensional data and consistently normalized nondimensional data. The procedure shown in this example should extend to any desired renormalization, provided the needed reference-state quantities are included in the database.

This example points out two stipulations that we now dictate for data in the class `NormalizedByUnknownDimensional`,

- (a) All nondimensional data within a given database that has `DataClass = NormalizedByUnknownDimensional` shall be consistently normalized.
- (b) Any nondimensional reference state appearing in a database should be sufficiently populated with reference quantities to allow for renormalization procedures.

The second of these stipulations is somewhat ambiguous, but good practice would suggest that a flow solver, for example, should output to the database enough static and/or stagnation reference quantities to sufficiently define the state.

Annex B shows an example of a well-populated reference state.

With these two stipulations, we contend the following:

- The dimensional scales used to nondimensionalize all data are immaterial, and there is no need to identify these quantities in a CGNS database.
- The dimensional scales need not be reference-state quantities provided in the database. For example, a given database could contain freestream reference state conditions, but all the data is normalized by sonic conditions (which are not provided in the database).
- All renormalization procedures can be carried out treating the data as if it were dimensional with a consistent set of units.
- Any application code that internally uses consistent normalization can use the data provided in a CGNS database without modification or transformation to the code's internal normalization.

Before ending this section, we note that the OVERFLOW flow solver mentioned above (or any other application code that internally uses inconsistent normalization) could easily read and write data to a nondimensional CGNS database that conforms to the above stipulations. On output, the code could renormalize data so it is consistently normalized. Probably, the easiest method would be to remove the molecular viscosity scale (μ_∞), and only use L , ρ_∞ and c_∞ for all normalizations

(recall these are dimensional scales). The only change from the above example would be the nondimensionalization of viscosity, which would become, $\tilde{\mu} = \mu/(\rho_{\infty}c_{\infty}L)$. The code could then output all field data as,

$$\begin{aligned}\tilde{x}_{ijk} &= x_{ijk}/L, & \tilde{u}_{ijk} &= u_{ijk}/c_\infty, & \tilde{\rho}_{ijk} &= \rho_{ijk}/\rho_\infty, \\ \tilde{y}_{ijk} &= y_{ijk}/L, & \tilde{v}_{ijk} &= v_{ijk}/c_\infty, & \tilde{p}_{ijk} &= p_{ijk}/(\rho_\infty c_\infty^2), \\ \tilde{z}_{ijk} &= z_{ijk}/L, & \tilde{w}_{ijk} &= w_{ijk}/c_\infty, & \tilde{\mu}_{ijk} &= \mu_{ijk}/(\rho_\infty c_\infty L),\end{aligned}$$

and output the freestream reference quantities,

$$\begin{aligned}\tilde{u}_\infty &= u_\infty/c_\infty, & \tilde{\rho}_\infty &= \rho_\infty/\rho_\infty = 1, \\ \tilde{v}_\infty &= v_\infty/c_\infty, & \tilde{p}_\infty &= p_\infty/(\rho_\infty c_\infty^2) = 1/\gamma, \\ \tilde{w}_\infty &= w_\infty/c_\infty, & \tilde{\mu}_\infty &= \mu_\infty/(\rho_\infty c_\infty L) \sim O(1/Re), \\ \tilde{c}_\infty &= c_\infty/c_\infty = 1, & \tilde{L} &= L/L = 1,\end{aligned}$$

where γ is the specific heat ratio (assumes a perfect gas) and Re is the Reynolds number.

On input, the flow solver should be able to recover its internal normalizations from the data in a nondimensional CGNS database by treating the data as if it were dimensional.

5.2.4 Nondimensional Parameters

If `DataClass = NondimensionalParameter`, the data is a nondimensional parameter (or array of nondimensional parameters). Examples include Mach number, Reynolds number and pressure coefficient. These parameters are prevalent in CFD, although their definitions tend to vary between different application codes. A list of standardized data-name identifiers for nondimensional parameters is provided in Annex A.4.

We distinguish nondimensional parameters from other data classes by the fact that they are *always* dimensionless. In a completely nondimensional database, they are distinct in that their normalization is not necessarily consistent with other data.

Typically, the `DimensionalUnits`, `DimensionalExponents` and `DataConversion` qualifiers are not used for nondimensional parameters; although, there are a few situations where they may be used (these are discussed below). Rather than rely on optional qualifiers to describe the normalization, we establish the convention that *any nondimensional parameters should be accompanied by their defining scales*; this is further discussed in Annex A.4. An example is Reynolds number defined as $Re = VL_R/\nu$, where V , L_R and ν are velocity, length, and viscosity scales, respectively. Note that these defining scales may be dimensional or nondimensional data. We establish the data-name identifiers `Reynolds`, `Reynolds_Velocity`, `Reynolds_Length` and `Reynolds_ViscosityKinematic` for the Reynolds number and its defining scales. Anywhere an instance of `DataArray_t` is found with the identifier `Reynolds`, there should also be entities for the defining scales. An example of this use for Reynolds number is given in Section 5.3.

In certain situations, it may be more convenient to use the optional qualifiers of `DataArray_t` to describe the normalization used in nondimensional parameters. These situations must satisfy two requirements: First, the defining scales are dimensional; and second, the nondimensional parameter is a normalization of a single ‘raw’ data quantity and it is clear what this raw data is. Examples

that satisfy this second constraint are pressure coefficient, where the raw data is static pressure, and lift coefficient, where the raw data is the lift force. Conversely, Reynolds number is a parameter that violates the second requirement—there are three pieces of raw data rather than one that make up Re . For nondimensional parameters that satisfy these two requirements, the qualifiers `DimensionalUnits`, `DimensionalExponents` and `DataConversion` may be used as in Section 5.2.2 to recover the raw dimensional data.

5.2.5 Dimensionless Constants

If `DataClass = DimensionlessConstant`, the data is a constant (or array of constants) with no associated dimensional units. The `DimensionalUnits`, `DimensionalExponents` and `DataConversion` qualifiers are not used.

5.3 Data-Array Examples

This section presents five examples of data-array entities and illustrates the use of optional information for describing dimensional and nondimensional data.

Example 5-A: One-Dimensional Data Array, Constants

A one-dimensional array of integers; the array is the integers from 1 to 10. The data is pure constants.

```
!  DataType = int
!  Dimension = 1
!  DimensionValues = 10
DataArray_t<int, 1, 10> Data1 =
  {{
    Data(int, 1, 10) = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] ;

    DataClass_t DataClass = DimensionlessConstant ;
  }} ;
```

The structure parameters for `DataArray_t` state the data is an one-dimensional integer array of length ten. The value of `DataClass` indicates the data is unitless constants.

Example 5-B: Two-Dimensional Data Array, Pressures

A two-dimensional array of pressures with size 11×9 given by the array $P(i,j)$. The data is dimensional with units of N/m^2 (i.e., $kg/(m \cdot s^2)$). Note that `Pressure` is the data-name identifier for static pressure.

```
!  DataType = real
!  Dimension = 2
!  DimensionValues = [11,9]
DataArray_t<real, 2, [11,9]> Pressure =
```

```

{{
Data(real, 2, [11,9]) = ((P(i,j), i=1,11), j=1,9) ;

DataClass_t DataClass = Dimensional ;

DimensionalUnits_t DimensionalUnits =
  {{
    MassUnits      = Kilogram ;
    LengthUnits    = Meter ;
    TimeUnits      = Second ;
    TemperatureUnits = Null ;
    AngleUnits     = Null ;
  }} ;

DimensionalExponents_t DimensionalExponents =
  {{
    MassExponent      = +1 ;
    LengthExponent    = -1 ;
    TimeExponent      = -2 ;
    TemperatureExponent = 0 ;
    AngleExponent     = 0 ;
  }} ;
}} ;

```

From the data-name identifier conventions presented in Annex A, **Pressure** has a floating-point data type; hence, the appropriate structure parameter for **DataArray_t** is **real**.

The value of **DataClass** indicates that the data is dimensional, and both the dimensional units and dimensional exponents are provided. **DimensionalUnits** specifies that the units are kilograms, meters, and seconds, and **DimensionalExponents** specifies the appropriate exponents for pressure. Combining the information gives pressure as $\text{kg}/(\text{m}\cdot\text{s}^2)$. **DimensionalExponents** could have been defaulted, since the dimensional exponents are given in Annex A for the data-name identifier **Pressure**.

Note that FORTRAN multidimensional array indexing is used to store the data; this is reflected in the FORTRAN-like implied do-loops for **P(i,j)**.

Example 5-C: Three-Dimensional Data Array, Nondimensional Static Enthalpy

A 3-D array of size $33 \times 9 \times 17$ containing nondimensional static enthalpy. The data is normalized by freestream velocity as follows:

$$\bar{h}_{i,j,k} = \frac{h_{i,j,k}}{q_{\text{ref}}^2},$$

where $\bar{h}_{i,j,k}$ is nondimensional static enthalpy. The freestream velocity is dimensional with a value of 10 m/s.

```
!  DataType = real
```

```

! Dimension = 3
! DimensionValues = [33,9,17]
DataArray_t<real, 3, [33,9,17]> Enthalpy =
  {{
    Data(real, 3, [33,9,17]) = ((H(i,j,k), i=1,33), j=1,9), k=1,17) ;

    DataClass_t DataClass = NormalizedByDimensional ;

    DataConversion_t DataConversion =
      {{
        real ConversionScale = 100 ;
        real ConversionOffset = 0 ;
      }} ;

    DimensionalUnits_t DimensionalUnits =
      {{
        MassUnits          = Null ;
        LengthUnits        = Meter ;
        TimeUnits           = Second ;
        TemperatureUnits    = Null ;
        AngleUnits          = Null ;
      }} ;

    DimensionalExponents_t DimensionalExponents =
      {{
        MassExponent        = 0 ;
        LengthExponent      = +2 ;
        TimeExponent        = -2 ;
        TemperatureExponent = 0 ;
        AngleExponent       = 0 ;
      }} ;
  }} ;

```

From Annex A, the identifier for static enthalpy is **Enthalpy** and its data type is **real**.

The value of **DataClass** indicates that the data is nondimensional and normalized by a dimensional reference quantity. **DataConversion** provides the conversion factors for recovering the raw static enthalpy, which has units of m^2/s^2 as indicated by **DimensionalUnits** and **DimensionalExponents**. Note that **DimensionalExponents** could have been defaulted using the conventions for the data-name identifier **Enthalpy**.

Example 5-D: Three-Dimensional Data Array, Nondimensional Database

The previous example for nondimensional enthalpy is repeated for a completely nondimensional database.

```
! DataType = real
```