Completing the CGNS Solution Description

Dr. Ken Alabi
TTC Technologies Inc.
New York.

Outline

- Convergence Data
- Flow Equations Description
- Reference States
- Dimensional information and units
- User Defined Data
- Examples

Convergence Data

- Flow solver convergence history information is described by the ConvergenceHistory_t structure. This structure contains the number of iterations and a list of data arrays containing convergence information at each iteration

Convergence Data

```
ConvergenceHistory_t :=
  {
  Descriptor_t NormDefinitions ;                                      (o)
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                  (o)

  int Iterations ;                                                    (r)

  List( DataArray_t<DataType, 1, Iterations>
    DataArray1 ... DataArrayN ) ;                                     (o)

  DataClass_t DataClass ;                                             (o)

  DimensionalUnits_t DimensionalUnits ;                               (o)

  List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;   (o)
  } ;
```

1. Default names for the Descriptor_t, DataArray_t, and UserDefinedData_t lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of ConvergenceHistory_t and shall not include the names DataClass, DimensionalUnits, or NormDefinitions.
2. Iterations is the only required field for ConvergenceHistory_t.
3. Iterations identifies the number of iterations for which convergence information is recorded. This value is also passed into each of the DataArray_t entities, defining the length of the data arrays.
4. DataClass defines the default for the class of data contained in the convergence history. If any convergence-history data is dimensional, DimensionalUnits may be used to describe the system of dimensional units employed. If present, these two entities take precedence of all corresponding entities at higher levels of the hierarchy, following the standard precedence rules.
5. The UserDefinedData_t data structure allows arbitrary user-defined data to be stored in Descriptor_t and DataArray_t children without the restrictions or implicit meanings imposed on these node types at other node locations.

Flow Equations Description

- The following types of governing equations can be described in a CGNS data file:

Governing Equation Type	SID Structure
Flow Equation Set Structure	FlowEquationSet_t
Governing Equations Structure	GoverningEquations_t
Thermodynamic Gas Model Structure	GasModel_t
Molecular Viscosity Model Structure	ViscosityModel_t
Thermal Conductivity Model Structure	ThermalConductivityModel_t
Turbulence Structure	
Thermal Relaxation Model Structure	ThermalRelaxationModel_t
Chemical Kinetics Model Structure	ChemicalKineticsModel_t
Electromagnetics Structure	

Flow Equations (cont'd)

```
FlowEquationSet_t< int CellDimension > :=
   {
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                    (o)

   int EquationDimension ;                                               (o)

   GoverningEquations_t<CellDimension> GoverningEquations ;              (o)

   GasModel_t GasModel ;                                                 (o)

   ViscosityModel_t ViscosityModel ;                                     (o)

   ThermalConductivityModel_t ThermalConductivityModel ;                 (o)

   TurbulenceClosure_t TurbulenceClosure ;                               (o)

   TurbulenceModel_t<CellDimension> TurbulenceModel ;                    (o)

   ThermalRelaxationModel_t ThermalRelaxationModel ;                     (o)

   ChemicalKineticsModel_t ChemicalKineticsModel ;                       (o)

   EMElectricFieldModel_t EMElectricFieldModel ;                         (o)

   EMMagneticFieldModel_t EMMagneticFieldModel ;                         (o)

   EMConductivityModel_t EMConductivityModel ;                           (o)

   DataClass_t DataClass ;                                               (o)

   DimensionalUnits_t DimensionalUnits ;                                 (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;   (o)
   } ;
```

Flow Equations (cont’d)

```
GoverningEquationsType_t := Enumeration(
   Null,
   FullPotential,
   Euler,
   NSLaminar,
   NSTurbulent,
   NSLaminarIncompressible,
   NSTurbulentIncompressible,
   UserDefined ) ;

 GoverningEquations_t< int CellDimension > :=
   {
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                  (o)

   GoverningEquationsType_t GoverningEquationsType ;                   (r)

   int[CellDimension*(CellDimension + 1)/2] DiffusionModel ;           (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;   (o)
   } ;
```

Flow Equations (cont'd)

```
GasModelType_t := Enumeration(
   Null,
   Ideal,
   VanderWaals,
   CaloricallyPerfect,
   ThermallyPerfect,
   ConstantDensity,
   RedlichKwong,
   UserDefined ) ;

 GasModel_t :=
   {
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                (o)

   GasModelType_t GasModelType ;                                      (r)

   List( DataArray_t<DataType, 1, 1> DataArray1 ... DataArrayN ) ;    (o)

   DataClass_t DataClass ;                                            (o)

   DimensionalUnits_t DimensionalUnits ;                              (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;  (o)
   } ;
```

Flow Equations (cont'd)

```
ViscosityModelType_t := Enumeration(
   Null,
   Constant,
   PowerLaw,
   SutherlandLaw,
   UserDefined ) ;

 ViscosityModel_t :=
   {
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                    (o)

   ViscosityModelType_t ViscosityModelType ;                             (r)

   List( DataArray_t<DataType, 1, 1> DataArray1 ... DataArrayN ) ;       (o)

   DataClass_t DataClass ;                                               (o)

   DimensionalUnits_t DimensionalUnits ;                                 (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;     (o)
   } ;
```

Flow Equations (cont'd)

```
ThermalConductivityModelType_t := Enumeration(
   Null,
   ConstantPrandtl,
   PowerLaw,
   SutherlandLaw,
   UserDefined ) ;

 ThermalConductivityModel_t :=
   {
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                 (o)

   ThermalConductivityModelType_t ThermalConductivityModelType ;      (r)

   List( DataArray_t<DataType, 1, 1> DataArray1 ... DataArrayN ) ;    (o)

   DataClass_t DataClass ;                                            (o)

   DimensionalUnits_t DimensionalUnits ;                              (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;  (o)
   } ;
```

Flow Equations (cont'd)

```
TurbulenceClosureType_t := Enumeration(
   Null,
   EddyViscosity,
   ReynoldsStress,
   ReynoldsStressAlgebraic,
   UserDefined ) ;

 TurbulenceClosure_t :=
   {
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                (o)

   TurbulenceClosureType_t TurbulenceClosureType ;                   (r)

   List( DataArray_t<DataType, 1, 1> DataArray1 ... DataArrayN ) ;   (o)

   DataClass_t DataClass ;                                           (o)

   DimensionalUnits_t DimensionalUnits ;                             (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ; (o)
   } ;
```

Flow Equations (cont'd)

```
TurbulenceModelType_t := Enumeration(
   Null,
   Algebraic_BaldwinLomax,
   Algebraic_CebeciSmith,
   HalfEquation_JohnsonKing,
   OneEquation_BaldwinBarth,
   OneEquation_SpalartAllmaras,
   TwoEquation_JonesLaunder,
   TwoEquation_MenterSST,
   TwoEquation_Wilcox,
   UserDefined ) ;

 TurbulenceModel_t< int CellDimension > :=
   {
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                (o)

   TurbulenceModelType_t TurbulenceModelType ;                        (r)

   List( DataArray_t<DataType, 1, 1> DataArray1 ... DataArrayN ) ;    (o)

   int[CellDimension*(CellDimension + 1)/2] DiffusionModel ;          (o)

   DataClass_t DataClass ;                                            (o)

   DimensionalUnits_t DimensionalUnits ;                              (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;  (o)
   } ;
```

Flow Equations (cont'd)

```
TurbulenceModelType_t := Enumeration(
   Null,
   Algebraic_BaldwinLomax,
   Algebraic_CebeciSmith,
   HalfEquation_JohnsonKing,
   OneEquation_BaldwinBarth,
   OneEquation_SpalartAllmaras,
   TwoEquation_JonesLaunder,
   TwoEquation_MenterSST,
   TwoEquation_Wilcox,
   UserDefined ) ;

 TurbulenceModel_t< int CellDimension > :=
   {
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                (o)

   TurbulenceModelType_t TurbulenceModelType ;                       (r)

   List( DataArray_t<DataType, 1, 1> DataArray1 ... DataArrayN ) ;   (o)

   int[CellDimension*(CellDimension + 1)/2] DiffusionModel ;         (o)

   DataClass_t DataClass ;                                           (o)

   DimensionalUnits_t DimensionalUnits ;                             (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ; (o)
   } ;
```

Flow Equations (cont'd)

Example - Spalart-Allmaras Turbulence Model

Description for the eddy-viscosity closure and Spalart-Allmaras turbulence model, including associated constants.

```
TurbulenceClosure_t TurbulenceClosure =
  {{
  TurbulenceClosureType_t TurbulenceClosureType = EddyViscosity ;

  DataArray_t<real, 1, 1> PrandtlTurbulent = {{ 0.90 }} ;
  }} ;

TurbulenceModel_t TurbulenceModel =
  {{
  TurbulenceModelType_t TurbulenceModelType = OneEquation_SpalartAllmaras ;

  DataArray_t<real, 1, 1> TurbulentSACb1   = {{ 0.1355 }} ;
  DataArray_t<real, 1, 1> TurbulentSACb2   = {{ 0.622 }} ;
  DataArray_t<real, 1, 1> TurbulentSASigma = {{ 2/3 }} ;
  DataArray_t<real, 1, 1> TurbulentSAKappa = {{ 0.41 }} ;
  DataArray_t<real, 1, 1> TurbulentSACw1   = {{ 3.2391 }} ;
  DataArray_t<real, 1, 1> TurbulentSACw2   = {{ 0.3 }} ;
  DataArray_t<real, 1, 1> TurbulentSACw3   = {{ 2 }} ;
  DataArray_t<real, 1, 1> TurbulentSACv1   = {{ 7.1 }} ;
  DataArray_t<real, 1, 1> TurbulentSACt1   = {{ 1 }} ;
  DataArray_t<real, 1, 1> TurbulentSACt2   = {{ 2 }} ;
  DataArray_t<real, 1, 1> TurbulentSACt3   = {{ 1.2 }} ;
  DataArray_t<real, 1, 1> TurbulentSACt4   = {{ 0.5 }} ;
  }} ;
```

Note that each DataArray_t entity is abbreviated.

Flow Equations (cont'd)

```
ThermalRelaxationModelType_t := Enumeration(
   Null,
   Frozen,
   ThermalEquilib,
   ThermalNonequilib,
   UserDefined ) ;

 ThermalRelaxationModel_t :=
   {
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                    (o)

   ThermalRelaxationModelType_t ThermalRelaxationModelType ;             (r)

   List( DataArray_t<DataType, 1, 1> DataArray1 ... DataArrayN ) ;       (o)

   DataClass_t DataClass ;                                               (o)

   DimensionalUnits_t DimensionalUnits ;                                 (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;     (o)
   } ;
```

Flow Equations (cont'd)

```
ChemicalKineticsModelType_t := Enumeration(
   Null,
   Frozen,
   ChemicalEquilibCurveFit,
   ChemicalEquilibMinimization,
   ChemicalNonequilib,
   UserDefined ) ;

 ChemicalKineticsModel_t :=
   {
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                   (o)

   ChemicalKineticsModelType_t ChemicalKineticsModelType ;              (r)

   List( DataArray_t<DataType, 1, 1> DataArray1 ... DataArrayN ) ;      (o)

   DataClass_t DataClass ;                                              (o)

   DimensionalUnits_t DimensionalUnits ;                                (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;    (o)
   } ;
```

Flow Equations (cont'd)

```
EMElectricFieldModelType_t := Enumeration(
   Null,
   Constant,
   Frozen,
   Interpolated,
   Voltage,
   UserDefined ) ;

 EMElectricFieldModel_t :=
   {
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                 (o)

   EMElectricFieldModelType_t EMElectricFieldModelType ;              (r)

   List( DataArray_t<DataType, 1, 1> DataArray1 ... DataArrayN ) ;    (o)

   DataClass_t DataClass ;                                            (o)

   DimensionalUnits_t DimensionalUnits ;                              (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;  (o)
   } ;
```

Flow Equations (cont'd)

```
EMConductivityModelType_t := Enumeration(
   Null,
   Constant,
   Frozen,
   Equilibrium_LinRessler,
   Chemistry_LinRessler,
   UserDefined ) ;

 EMConductivityModel_t :=
   {
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                (o)

   EMConductivityModelType_t EMConductivityModelType ;               (r)

   List( DataArray_t<DataType, 1, 1> DataArray1 ... DataArrayN ) ;   (o)

   DataClass_t DataClass ;                                           (o)

   DimensionalUnits_t DimensionalUnits ;                             (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ; (o)
   } ;
```

Flow Equations (cont'd)

```
FlowEquationSet_t<3> EulerEquations =
   {{
   int EquationDimension = 3 ;

   GoverningEquations_t<3> GoverningEquations =
     {{
     GoverningEquationsType_t GoverningEquationsType = Euler ;
     }} ;

   GasModel_t GasModel =
     {{
     GasModelType_t GasModelType = CaloricallyPerfect ;

     DataArray_t<real, 1, 1> SpecificHeatRatio =
       {{
       Data(real, 1, 1) = 1.667 ;

       DataClass_t DataClass = NondimensionalParameter ;
       }} ;
     }} ;
   }} ;
```

Flow Equations (cont'd)

```
FlowEquationSet_t<3> NSEquations =
   {{
   int EquationDimension = 3 ;

   GoverningEquations_t<3> GoverningEquations =
     {{
     GoverningEquationsType_t GoverningEquationsType = NSTurbulent ;

     int[6] DiffusionModel = [1,1,1,1,1,1] ;
     }} ;

   GasModel_t GasModel =
     {{
     GasModelType_t GasModelType = CaloricallyPerfect ;

     DataArray_t<real, 1, 1> SpecificHeatRatio = {{ 1.4 }} ;
     }} ;

   ViscosityModel_t ViscosityModel =
     {{
     ViscosityModelType_t ViscosityModelType = SutherlandLaw ;

     DataArray_t<real, 1, 1> SutherlandLawConstant =
       {{
       Data(real, 1, 1) = 110.6 }} ;

       DataClass_t DataClass = Dimensional ;
       DimensionalUnits_t DimensionalUnits = {{ TemperatureUnits = Kelvin }} ;
       }} ;
     }} ;
```

Flow Equations (cont'd)

```
ThermalConductivityModel_t ThermalConductivityModel =
  {{
  ThermalConductivityModelType_t ThermalConductivityModelType =
     ConstantPrandtl ;

  DataArray_t<real, 1, 1> Prandtl = {{ 0.72 }} ;
  }} ;

TurbulenceClosure_t<3> TurbulenceClosure =
  {{
  TurbulenceClosureType_t TurbulenceClosureType = EddyViscosity ;

  DataArray_t<real, 1, 1> PrandtlTurbulent = {{ 0.90 }} ;
  }} ;

TurbulenceModel_t<3> TurbulenceModel =
  {{
  TurbulenceModelType_t TurbulenceModelType = OneEquation_SpalartAllmaras ;

  int[6] DiffusionModel = [1,1,1,1,1,1] ;
  }} ;
}} ;
```

Reference States

- Reference state is a list of geometric or flow-state quantities defined at a common location or condition.
- Described by The ReferenceState_t structure
- Examples of typical reference states associated with CFD calculations are freestream, plenum, stagnation, inlet and exit. Note that providing a ReferenceState description is particularly important if items elsewhere in the CGNS database are NormalizedByUnknownDimensional.

Reference States (cont'd)

```
ReferenceState_t :=
   {
   Descriptor_t ReferenceStateDescription ;                          (o)
   List( Descriptor_t Descriptor1 ... DescriptorN ) ;                (o)

   List( DataArray_t<DataType, 1, 1> DataArray1 ... DataArrayN ) ;   (o)

   DataClass_t DataClass ;                                           (o)

   DimensionalUnits_t DimensionalUnits ;                             (o)

   List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ; (o)
   } ;
```

Reference States (cont'd)

Data-name Identifiers for Reference State

Data-Name Identifier	Description	Units
`Mach`	Mach number, $M = q/c$	-
`Mach_Velocity`	Velocity scale, q	**L/T**
`Mach_VelocitySound`	Speed of sound scale, c	**L/T**
`Reynolds`	Reynolds number, $Re = VL_R / \nu$	-
`Reynolds_Velocity`	Velocity scale, V	**L/T**
`Reynolds_Length`	Length scale, L_R	**L**
`Reynolds_ViscosityKinematic`	Kinematic viscosity scale, ν	$\mathbf{L^2/T}$
`LengthReference`	Reference length, L	**L**

- In addition, any flowfield quantities (such as Density, Pressure, etc.) can be included in the ReferenceState.
- The reference length *L* (LengthReference) may be necessary for NormalizedByUnknownDimensional databases, to define the length scale used for nondimensionalizations. It may be the same or different from the Reynolds_Length used to define the Reynolds number.
- Because of different definitions for angle of attack and angle of yaw, these quantities are not explicitly defined in the SIDS. Instead, the user can unambigouosly denote the freestream velocity vector direction by giving VelocityX, VelocityY, and VelocityZ in ReferenceState, (with the reference state denoting the freestream).

Reference States (cont'd)

- **Example - Reference State with Dimensional Data**
- A freestream reference state where all data quantities are dimensional. Standard atmospheric conditions at sea level are assumed for static quantities, and all stagnation variables are obtained using the isentropic relations. The flow velocity is 200 m/s aligned with the *x*-axis. Dimensional units of kilograms, meters, and seconds are used. The data class and system of units are specified at the ReferenceState_t level rather than attaching this information directly to the DataArray_t entities for each reference quantity. Data-name identifiers are provided in the section Conventions for Data-Name Identifiers.

Reference States (cont'd)

- **Example - Reference State with Dimensional Data**

```
ReferenceState_t ReferenceState =
   {{
   Descriptor_t ReferenceStateDescription =
     {{
     Data(char, 1, 45) = "Freestream at standard atmospheric conditions" ;
     }} ;

   DataClass_t DataClass = Dimensional ;

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

   DataArray_t<real, 1, 1> VelocityX =
        {{
        Data(real, 1, 1) = 200. ;
        }} ;
   DataArray_t<real, 1, 1> VelocityY              = {{ 0. }} ;
   DataArray_t<real, 1, 1> VelocityZ              = {{ 0. }} ;

   DataArray_t<real, 1, 1> Pressure               = {{ 1.0132E+05 }} ;
   DataArray_t<real, 1, 1> Density                = {{ 1.226 }} ;
   DataArray_t<real, 1, 1> Temperature            = {{ 288.15 }} ;
   DataArray_t<real, 1, 1> VelocitySound          = {{ 340. }} ;
   DataArray_t<real, 1, 1> ViscosityMolecular     = {{ 1.780E-05 }} ;

   DataArray_t<real, 1, 1> PressureStagnation     = {{ 1.2806E+05 }} ;
   DataArray_t<real, 1, 1> DensityStagnation      = {{ 1.449 }} ;
   DataArray_t<real, 1, 1> TemperatureStagnation  = {{ 308.09 }} ;
   DataArray_t<real, 1, 1> VelocitySoundStagnation = {{ 351.6 }} ;

   DataArray_t<real, 1, 1> PressureDynamic        = {{ 0.2542E+05 }} ;
   }} ;
```

Dimensional Information and Units

- Dimensional information 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.

Basic Dimensional Unit	**Enumeration**
MassUnits_t	Null, Kilogram, Gram, Slug,PoundMass, UserDefined
LengthUnits_t	Null, Meter, Centimeter, Millimeter, Foot, Inch, UserDefined
TimeUnits_t	Null, Second, UserDefined
TemperatureUnits_t	Null, Kelvin, Celsius, Rankine, Fahrenheit, UserDefined
AngleUnits_t	Null, Degree, Radian, UserDefined
ElectricCurrentUnits_t	Null, Ampere, Abampere, Statampere, Edison, auCurrent, UserDefined
SubstanceAmountUnits_t	Null, Mole, Entities, StandardCubicFoot, StandardCubicMeter, UserDefined
LuminousIntensityUnits_t	Null, Candela, Candle, Carcel, Hefner, Violle, UserDefined

Dimensional Information and Units (cont'd)

- Dimensional information is contained in a `DimensionalUnits_t` node

```
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)
    }
  } ;
```

Dimensional Information and Units (cont'd)

- 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

Dimensional Information and Units (cont'd)

- `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)
   }
 } ;
```

- .

Dimensional Information and Units (cont'd)

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

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

Dimensional Information and Units (cont’d)

- 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:

```
Data(raw) = Data(nondimensional)*ConversionScale + ConversionOffset
```

Dimensional Information and Units (cont'd)

- Example: 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/m2 (i.e., kg/(m-s2)). Note that Pressure is the data-name identifier for static pressure.

```
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 ;
    }} ;
  }} ;
```

Dimensional Information and Units (cont'd)

- 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:

```
Data(raw) = Data(nondimensional)*ConversionScale + ConversionOffset
```

- If DataClass = NormalizedByDimensional, the data is nondimensional and is normalized by dimensional reference quantities.

User Defined Data

- UserDefinedData_t provides a means of storing 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.
- Several nodes accommodate user defined data including:
  - GridCoordinates_t
  - FlowSolution_t

User Defined Data (cont'd)

```
UserDefinedData_t :=
  {
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                  (o)

  GridLocation_t GridLocation ;                                       (o/d)

  IndexRange_t<IndexDimension> PointRange ;                           (o)
  IndexArray_t<IndexDimension, ListLength, int> PointList ;           (o)

  List( DataArray_t<> DataArray1 ... DataArrayN ) ;                   (o)

  DataClass_t DataClass ;                                             (o)

  DimensionalUnits_t DimensionalUnits ;                               (o)

  FamilyName_t FamilyName ;                                           (o)

  List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;   (o)

  int Ordinal ;                                                       (o)
  } ;
```

User Defined Data (cont'd)

- UserDefinedData_t data structure:
  - Default names for the Descriptor_t, DataArray_t, and UserDefinedData_t lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of UserDefinedData_t and shall not include the names DataClass, DimensionalUnits, FamilyName, GridLocation, Ordinal, PointList, or PointRange.
  - GridLocation may be set to Vertex, IFaceCenter, JFaceCenter, KFaceCenter, FaceCenter, CellCenter, or EdgeCenter. If GridLocation is absent, then its default value is Vertex. When GridLocation is set to Vertex, then PointList or PointRange refer to node indices, for both structured and unstructured grids. When GridLocation is set to FaceCenter, then PointList or PointRange refer to face elements.
  - GridLocation, PointRange, and PointList may only be used when UserDefinedData_t is located below a Zone_t structure in the database hierarchy.
  - Only one of PointRange and PointList may be specified.