CGNS Proposal for Extension 0050

Solution Continuity Type

Version 1.0

April 22, 2026

CPEX#	0050
Scope	Distinguish continuous and discontinuous solution continuity
Champion	M. Scot Breitenfeld
Organization	The HDF Group
E-mail	brtnfld@hdfgroup.org
GitHub Issue	TBD
Reference Impl.	TBD
Target Release	TBD
Date First Posted	April 22, 2026
SIDS Status	proposed
Filemap Status	proposed
MLL Status	proposed

Contents

1 Abstract

This CPEX introduces a `SolutionContinuity_t` enumeration and corresponding SIDS node to the CGNS standard, thereby enabling CGNS files to specify whether solution data exhibits continuous or discontinuous field continuity. This distinction, most commonly associated with Continuous Galerkin (CG) and Discontinuous Galerkin (DG) finite element methods, is essential for the correct interpretation of solution data, yet is not addressed by any existing CGNS construct. The proposal is broadly applicable: it is valid for any polynomial order, any discretization family (including finite element, finite volume, and spectral element methods), and integrates seamlessly with the higher-order solution infrastructure introduced by CPEX 0045.

2 Motivation and Rationale

2.1 Problem Statement

Modern CFD and computational mechanics codes increasingly use discretization methods whose solution data cannot be fully described by the existing CGNS metadata. The core issue is the treatment of degrees of freedom (DOFs) at shared geometric entities (vertices, edges, faces):

- In a **continuous** representation (e.g., Continuous Galerkin FEM, C^0 spectral element methods), DOFs at shared interfaces are *shared*: a single value exists per geometric node, and the global solution field is at least C^0-continuous across element boundaries.
- In a **discontinuous** representation (e.g., Discontinuous Galerkin, finite volume, C^{-1} spectral element methods), DOFs are *element-local*: each element owns its own values at every point, including at shared interfaces. Multiple independent values may exist at the same geometric location.

This distinction affects:

1. **Data size and layout.** A DG solution on a mesh with shared vertices stores more values than a CG solution on the same mesh. Post-processors that assume shared-vertex semantics will misinterpret the data dimensions.
2. **Visualization and interpolation.** CG fields can be interpolated globally using the mesh connectivity; DG fields require element-local interpolation and may exhibit intentional discontinuities at element interfaces that should not be smoothed.
3. **Restart and data exchange.** Solvers reading restart files must know whether DOFs are shared or element-local to correctly reconstruct the solution state. Coupling frameworks exchanging solution data between codes need this metadata to perform correct inter-code transfers. In Fluid-Structure Interaction (FSI) applications, for example, a continuous FEM structural displacement field and a discontinuous DG fluid field coexist—the coupling framework must use different interpolation and projection operators for each, and the continuity metadata enables this dispatch.
4. **Error estimation and adaptation.** The jump in solution values across element interfaces in a DG solution is physically meaningful and is used for error estimation. A post-processor that incorrectly treats DG data as continuous would average out these jumps.

2.2 Inadequacy of Existing CGNS Constructs

No existing CGNS mechanism captures the continuous/discontinuous distinction:

`GridLocation_t` specifies *where* data is located (vertices, cell centers, faces, edges, or interpolation points), but not whether values at shared locations are shared or element-local. A `Vertex`-located solution could be either CG or DG; the same applies to `InterpolationPoints`.

`PointSet` (`PointList`, `PointRange`) can index subsets of data but does not encode continuity semantics.

`InterpolationType_t` (CPEX 0045) describes the polynomial basis (e.g., Lagrange, monomial), but not whether that basis yields a globally continuous or element-local field.

`SolutionInterpolation_t` (CPEX 0045) records polynomial order and basis type per element type but contains no continuity information.

Data array size can sometimes be used to *infer* the representation—a DG field will have $N_{\text{elem}} \times N_{\text{DOF/elem}}$ values versus N_{nodes} for CG. However, this inference is fragile, ambiguous for $p = 0$ (cell-centered) data, and breaks entirely for mixed-element meshes or when point sets are used.

2.3 Generality Beyond Higher-Order Methods

Although often discussed in the context of high-order finite elements, the continuous/discontinuous distinction is **not limited to higher-order methods**:

- **DG** $p = 0$ (piecewise constant) is essentially a cell-centered finite volume representation.
- **DG** $p = 1$ (piecewise linear, discontinuous) is widely used and is a low-order method.
- **CG** $p = 1$ is standard continuous linear FEM, the most common finite element discretization.
- Classical **finite volume** representations are inherently discontinuous ($p = 0$ DG).
- **Hybridizable DG (HDG)** methods have both element-local DOFs (discontinuous) and trace DOFs on faces (continuous on the skeleton).
- **FSI and multi-physics coupling** often combines a continuous FEM structural solver (shared displacement DOFs) with a discontinuous DG or FV fluid solver (element-local flow variables) within the same file or across coupled zones.

Accordingly, a proper metadata field should therefore live at the `FlowSolution_t` level, applicable at any order, not solely within the higher-order interpolation infrastructure.

3 Detailed Description

3.1 Design Overview

The proposal introduces:

1. A new `SolutionContinuity_t` enumeration with values covering the common continuity types.
2. A new optional child node of `FlowSolution_t`, `DiscreteData_t`, and `ZoneSubRegion_t` recording the solution continuity.
3. Mid-Level Library functions to read and write the continuity.

4. Fortran bindings.

3.2 New Enumeration: `SolutionContinuity_t`

```
typedef enum {
  CGNS_ENUMV( SolutionContinuityNull )        = CG_Null,
  CGNS_ENUMV( SolutionContinuityUserDefined )  = CG_UserDefined,
  CGNS_ENUMV( SolutionContinuous )                = 2,
  CGNS_ENUMV( SolutionDiscontinuous )             = 3
} CGNS_ENUMT( SolutionContinuity_t );

#define NofValidSolutionContinuity 4

extern CGNSDLL const char *
    SolutionContinuityName[NofValidSolutionContinuity];

CGNSDLL const char *cg_SolutionContinuityName(
    CGNS_ENUMT(SolutionContinuity_t) type);
```

The enumeration values are:

`SolutionContinuityNull` No continuity specified. This is the default for backward compatibility: existing files and codes that do not set this field are unaffected.

`SolutionContinuityUserDefined` Application-defined continuity not covered by the standard values. When this value is used, writers *should* attach a `Descriptor_t` child node named `"ContinuityDescription"` to the parent `FlowSolution_t`, `DiscreteData_t`, or `ZoneSubRegion_t` node. The string value should identify the continuity convention (e.g., `"HDG-trace"`, `"C1-spline"`). This allows readers to recognize the convention without requiring a separate side-channel.

`SolutionContinuous` DOFs at shared mesh entities (vertices, edges, faces) are *shared* between adjacent elements: each shared entity carries exactly one value. This is a statement about *storage and DOF-sharing semantics*, not about the mathematical smoothness of the field. Consequently, $\boldsymbol{H}(\mathrm{div})$-conforming spaces (e.g., Raviart–Thomas) and $\boldsymbol{H}(\mathrm{curl})$-conforming spaces (e.g., Nédélec) should also use this label: their DOFs are shared at element interfaces (normal-flux or tangential-component DOFs, respectively), even though the full vector field is not pointwise C^0-continuous. Typical of: Continuous Galerkin FEM, C^0 spectral element methods, isogeometric analysis with C^0 junctions, $\boldsymbol{H}(\mathrm{div})$ and $\boldsymbol{H}(\mathrm{curl})$ conforming spaces.

`SolutionDiscontinuous` The solution field is element-local: each element owns independent DOFs, including at shared geometric locations, and no interface DOF is shared between adjacent elements. Typical of: Discontinuous Galerkin methods, finite volume methods, C^{-1} spectral element methods.

3.2.1 Rationale for Two-Value Classification

The binary continuous/discontinuous classification (enum values `SolutionContinuous` and `SolutionDiscontinuous`) is deliberate. Methods that appear to require finer distinctions are already handled by the existing two values, combined with standard CGNS patterns:

HDG and mixed-continuity methods use both element-local volume DOFs and globally continuous trace DOFs. These are stored as separate `FlowSolution_t` nodes—one marked `SolutionDiscontinuous`, the other `SolutionContinuous`—following the same pattern

CGNS already uses to separate fields at different `GridLocation_t` values. A dedicated `MixedContinuity` value would conflate two distinct data layouts into a single node, making it harder, not easier, for post-processors to interpret the data.

Penalized / weakly-continuous DG methods enforce continuity through penalty terms in the variational formulation, but the data layout remains element-local: each element owns its own DOFs. The correct label is `SolutionDiscontinuous`. "Weak continuity" is a solver-internal property of the numerical method, not a property of the stored data that readers and post-processors need to act on.

Isogeometric analysis (C^k, $k \geq 1$) uses basis functions with higher-than-C^0 smoothness, but the data layout is still globally continuous with shared DOFs at patch interfaces. The correct label is `SolutionContinuous`. The smoothness order k is a property of the basis, not of the data layout; it belongs in the interpolation metadata (CPEX 0045), not in the continuity type.

The `SolutionContinuityUserDefined` value remains available as an escape hatch, consistent with CGNS convention for all enumeration types.

Scope note. `SolutionContinuity_t` captures *storage and DOF-sharing semantics* only—it answers "are interface DOFs shared or element-local?" It does not encode the polynomial basis, smoothness order, patch topology, or support-radius metadata that isogeometric analysis (IGA), meshless, or other non-standard discretizations may additionally require. Those richer descriptions belong in `SolutionInterpolation_t` (CPEX 0045) or in future extensions; this CPEX is intentionally scoped to the coarser, universally applicable continuous/discontinuous distinction.

3.3 SIDS Node Definition

The continuity is stored as a new optional child node of `FlowSolution_t`. The amended SIDS definition adds one line (shown with the `+` marker). The definition below includes the `SolutionInterpolation_t` child introduced by CPEX 0045:

`FlowSolution_t (amended, SIDS Section 7.7)`

```
FlowSolution_t< int CellDimension, int IndexDimension,
int VertexSize[IndexDimension],
int CellSize[IndexDimension] > :=
{
List( Descriptor_t Descriptor1 ...  DescriptorN ) ;          (o)

GridLocation_t GridLocation ;                              (o/d)

Rind_t<IndexDimension> Rind ;                             (o/d)

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

List( DataArray_t<DataType, IndexDimension, DataSize[]>
DataArray1 ...  DataArrayN ) ;                            (o)
```

```
DataClass_t DataClass ;                                    (o)

DimensionalUnits_t DimensionalUnits ;                      (o)

SolutionInterpolation_t SolutionInterpolation ;            (o)    (CPEX
0045)

+   SolutionContinuity_t SolutionContinuity ;              (o)

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

The `SolutionContinuity` child is **optional** (`o`). When absent, the continuity is unspecified (`SolutionContinuityNull`), preserving full backward compatibility.

The same optional child node is added to `DiscreteData_t` (SIDS Section 12.4), using identical structure and semantics:

`DiscreteData_t (amended, SIDS Section 12.4)`

```
DiscreteData_t< int CellDimension, int IndexDimension,
int VertexSize[IndexDimension],
int CellSize[IndexDimension] > :=
{
...  (existing children unchanged) ...

+   SolutionContinuity_t SolutionContinuity ;              (o)
}
```

The optional child node is also added to `ZoneSubRegion_t` (SIDS Section 7.9). `ZoneSubRegion_t` holds `DataArray_t` nodes for localized solutions, boundary fluxes, or fields on sub-regions where the continuous/discontinuous distinction applies equally:

`ZoneSubRegion_t (amended, SIDS Section 7.9)`

```
ZoneSubRegion_t< int RegionCellDimension > :=
{
...  (existing children unchanged) ...

+   SolutionContinuity_t SolutionContinuity ;              (o)
}
```

3.3.1 Node Placement Rationale

The node is placed as a child of `FlowSolution_t` (rather than, for example, `Zone_t` or `SolutionInterpolation_t`) because:

- The continuity is a property of the *solution field*, not of the mesh or the zone.
- A single zone may contain multiple `FlowSolution_t` nodes with different continuity—for example, an HDG solver storing both volume (discontinuous) and trace (continuous) solutions, or a multi-physics code storing a continuous displacement field alongside a discontinuous stress field.
- This parallels the existing `GridLocation_t` child of `FlowSolution_t`, which is also a per-solution-node property.
- Placing it within the CPEX 0045 `SolutionInterpolation_t` would incorrectly limit the scope to higher-order solutions, when the distinction applies at any order.

3.4 New MLL Functions

3.4.1 C API

```c
/*
 * Write the solution continuity for a FlowSolution_t node.
 *
 * fn   - file number
 * B    - base index
 * Z    - zone index
 * S    - solution index
 * type - SolutionContinuity_t value
 */
int cg_sol_continuity_write(
    int fn, int B, int Z, int S,
    CGNS_ENUMT(SolutionContinuity_t) type);

/*
 * Read the solution continuity for a FlowSolution_t node.
 *
 * fn   - file number
 * B    - base index
 * Z    - zone index
 * S    - solution index
 * type - [out] SolutionContinuity_t value;
 *        returns SolutionContinuityNull if node absent
 */
int cg_sol_continuity_read(
    int fn, int B, int Z, int S,
    CGNS_ENUMT(SolutionContinuity_t) *type);

/*
 * Write/read the solution continuity for a DiscreteData_t node.
 * Same semantics as the FlowSolution_t counterparts.
 *
 * fn   - file number
 * B    - base index
 * Z    - zone index
 * D    - discrete data index
 * type - SolutionContinuity_t value
 */
int cg_discrete_continuity_write(
    int fn, int B, int Z, int D,
    CGNS_ENUMT(SolutionContinuity_t) type);

int cg_discrete_continuity_read(
    int fn, int B, int Z, int D,
    CGNS_ENUMT(SolutionContinuity_t) *type);

/*
 * Write/read the solution continuity for a ZoneSubRegion_t node.
```

```c
 * Same semantics as the FlowSolution_t counterparts.
 *
 * fn    - file number
 * B     - base index
 * Z     - zone index
 * SR    - sub-region index
 * type - SolutionContinuity_t value
 */
int cg_subreg_continuity_write(
    int fn, int B, int Z, int SR,
    CGNS_ENUMT(SolutionContinuity_t) type);

int cg_subreg_continuity_read(
    int fn, int B, int Z, int SR,
    CGNS_ENUMT(SolutionContinuity_t) *type);

/*
 * Return the string name for a SolutionContinuity_t value.
 * Matches the convention of cg_GridLocationName, cg_DataTypeName, etc.
 * Returns NULL for out-of-range values.
 */
CGNSDLL const char *cg_SolutionContinuityName(
    CGNS_ENUMT(SolutionContinuity_t) type);
```

Behavior:

- The `_write` functions create (or overwrite) the `SolutionContinuity` child node under the specified `FlowSolution_t`, `DiscreteData_t`, or `ZoneSubRegion_t` node. Writing `SolutionContinuityNull` removes the node if present.
- The `_read` functions return the value stored in the `SolutionContinuity` child node. If the node does not exist (legacy file or unset), they return `SolutionContinuityNull` without error.

3.4.2 Fortran API

```
! FlowSolution_t
subroutine cg_sol_continuity_write_f( &
    fn, B, Z, S, continuity, ier)
  integer, intent(in) :: fn, B, Z, S, continuity
  integer, intent(out) :: ier

subroutine cg_sol_continuity_read_f( &
    fn, B, Z, S, continuity, ier)
  integer, intent(in) :: fn, B, Z, S
  integer, intent(out) :: continuity, ier

! DiscreteData_t
subroutine cg_discrete_continuity_write_f( &
    fn, B, Z, D, continuity, ier)
  integer, intent(in) :: fn, B, Z, D, continuity
  integer, intent(out) :: ier

subroutine cg_discrete_continuity_read_f( &
    fn, B, Z, D, continuity, ier)
  integer, intent(in) :: fn, B, Z, D
  integer, intent(out) :: continuity, ier

! ZoneSubRegion_t
subroutine cg_subreg_continuity_write_f( &
    fn, B, Z, SR, continuity, ier)
  integer, intent(in) :: fn, B, Z, SR, continuity
  integer, intent(out) :: ier

subroutine cg_subreg_continuity_read_f( &
```

```
    fn, B, Z, SR, continuity, ier)
  integer, intent(in) :: fn, B, Z, SR
  integer, intent(out) :: continuity, ier
```

3.5 Internal Data Structure Changes

The `cgns_sol` structure (`FlowSolution_t`), `cgns_discrete` structure (`DiscreteData_t`), and `cgns_subreg` structure (`ZoneSubRegion_t`) in `cgns_header.h` each gain two fields:

```
typedef struct {            /* FlowSolution_t node */
    /* ... existing fields ... */

    /* CPEX 0050 */
    CGNS_ENUMT(SolutionContinuity_t) continuity;
    double continuity_id;   /* CGIO node id for overwrite/delete */
} cgns_sol;

typedef struct {            /* DiscreteData_t node */
    /* ... existing fields ... */

    /* CPEX 0050 */
    CGNS_ENUMT(SolutionContinuity_t) continuity;
    double continuity_id;
} cgns_discrete;

typedef struct {            /* ZoneSubRegion_t node */
    /* ... existing fields ... */

    /* CPEX 0050 */
    CGNS_ENUMT(SolutionContinuity_t) continuity;
    double continuity_id;
} cgns_subreg;
```

The `continuity` field defaults to `SolutionContinuityNull` (value 0), which is the zero-initialization value and the "not specified" state. The `continuity_id` field stores the CGIO node identifier so that the write function can overwrite or delete the child node; it defaults to 0 (no node).

3.6 Internal Lookup Table Initialization

The string-to-enum mapping table must be initialized in `cgns_internals.c` so that `cgi_SolutionContinuity()` can resolve the C1 character data stored in CGNS files:

```
const char *SolutionContinuityName[NofValidSolutionContinuity] = {
    "Null",
    "UserDefined",
    "Continuous",
    "Discontinuous"
};
```

3.7 MLL Implementation Requirements

3.7.1 Internal Tree Parser Update (`cgns_internals.c`)

When a CGNS file is opened, the MLL tree parser crawls the HDF5/CGIO tree and populates internal C structures. The parser must be updated to recognize child nodes with label `SolutionContinuity_t` during the traversal of `FlowSolution_t` (`cgi_read_sol`), `DiscreteData_t` (`cgi_read_discrete`), and `ZoneSubRegion_t` (`cgi_read_subregion`).

The required behavior is:

1. During memory allocation of `cgns_sol`, `cgns_discrete`, and `cgns_subreg`, the `continuity` field is zero-initialized, which corresponds to `SolutionContinuityNull`.
2. During the child-node traversal, if a child with label `"SolutionContinuity_t"` is found, read its C1 string data and map it to the enum value via `cgi_SolutionContinuity(string_data, &continuity)`. Store the result in the struct's `continuity` field.
3. If no such child node exists, the field retains its default `SolutionContinuityNull` value.

This follows the existing pattern used for `GridLocation_t` parsing in `cgi_read_sol`.

3.7.2 Read Function Error Handling

The `_read` functions (e.g., `cg_sol_continuity_read`) return the cached struct field rather than performing a fresh HDF5 read. Because the internal tree parser already handles the absent-node case by leaving the field at `SolutionContinuityNull`, the read function simply returns the struct value and `CG_OK`:

```
int cg_sol_continuity_read(int fn, int B, int Z, int S,
                           CGNS_ENUMT(SolutionContinuity_t) *type) {
    cgns_sol *sol;

    cg = cgi_get_file(fn);
    if (cg == 0) return CG_ERROR;

    if (cgi_check_mode(cg->filename, cg->mode, CG_MODE_READ))
        return CG_ERROR;

    sol = cgi_get_sol(cg, B, Z, S);
    if (sol == NULL) return CG_ERROR;

    *type = sol->continuity;

    return CG_OK;
}
```

This architecture avoids the need to intercept `CG_NODE_NOT_FOUND` errors at read time—the absent-node case is handled once during file open, and all subsequent reads are served from the in-memory cache. Legacy code that iterates over solutions and queries the continuity will always receive `CG_OK` with `SolutionContinuityNull` for files that predate this CPEX.

3.7.3 Write Function Implementation

The `_write` functions create (or overwrite) the `SolutionContinuity` child node. Writing `SolutionContinuityNull` removes the node if present, restoring the "unspecified" state. The reference implementation for `FlowSolution_t` is shown below; the `DiscreteData_t` and `ZoneSubRegion_t` variants are analogous (substituting the appropriate internal structure accessor).

```c
int cg_sol_continuity_write(int fn, int B, int Z, int S,
                            CGNS_ENUMT(SolutionContinuity_t) type) {
    cgns_sol *sol;
    cgsize_t length;
    double posit_id;

    cg = cgi_get_file(fn);
    if (cg == 0) return CG_ERROR;

    if (cgi_check_mode(cg->filename, cg->mode, CG_MODE_WRITE))
        return CG_ERROR;

    sol = cgi_get_sol(cg, B, Z, S);
    if (sol == NULL) return CG_ERROR;

    /* Validate enum value */
    switch (type) {
        case CGNS_ENUMV(SolutionContinuityNull):
        case CGNS_ENUMV(SolutionContinuityUserDefined):
        case CGNS_ENUMV(SolutionContinuous):
        case CGNS_ENUMV(SolutionDiscontinuous):
            break;
        default:
            cgi_error("Invalid SolutionContinuity value: %d", (int)type);
            return CG_ERROR;
    }

    /* Delete existing node if writing Null */
    if (type == CGNS_ENUMV(SolutionContinuityNull)) {
        if (sol->continuity !=
                CGNS_ENUMV(SolutionContinuityNull)) {
            if (cgi_delete_node(sol->id,
                    sol->continuity_id))
                return CG_ERROR;
            sol->continuity =
                CGNS_ENUMV(SolutionContinuityNull);
            sol->continuity_id = 0;
        }
        return CG_OK;
    }

    /* Delete existing node before overwrite */
    if (sol->continuity !=
            CGNS_ENUMV(SolutionContinuityNull)) {
        if (cgi_delete_node(sol->id,
                sol->continuity_id))
            return CG_ERROR;
        sol->continuity_id = 0;
    }

    /* Create the child node */
    length = (cgsize_t)strlen(
        SolutionContinuityName[type]);
    if (cgi_new_node(sol->id, "SolutionContinuity",
            "SolutionContinuity_t", &posit_id,
            "C1", 1, &length,
            (void *)SolutionContinuityName[type]))
        return CG_ERROR;

    sol->continuity = type;
    sol->continuity_id = posit_id;
    return CG_OK;
}
```

Note: The `continuity_id` field (a `double` storing the CGIO node identifier) must be added to the

internal structures alongside the `continuity` field so that the node can be located for overwrite or deletion. This follows the pattern used by other optional child nodes in the MLL (e.g., `GridLocation` tracking via `location_id` fields in some internal paths).

3.8 Data Layout for Discontinuous Solutions at Shared Mesh Vertices

A common and important case arises when a DG solver uses a mesh whose vertices are shared in the standard CGNS sense—enabling mesh walking, stream tracing, and connectivity queries via the standard `Elements_t` tables—but stores element-local, independent solution values at each element's copy of those vertices. At a shared vertex adjacent to k elements there are k distinct solution values, one per element; readers must not average or conflate them.

The correct CGNS encoding for this pattern is `GridLocation_t = Vertex` combined with `SolutionContinuity_t = SolutionDiscontinuous` and a `PointList` that lists every element-vertex pair explicitly (repeated vertex indices are expected and meaningful). The `SolutionContinuity_t` node is the machine-readable flag that tells readers these repeated entries represent independent DOFs, not duplicate data.

Without the `SolutionContinuity_t` node, a reader cannot distinguish this layout from a malformed continuous solution with duplicate entries, which is why the node is necessary. With it, a post-processor or coupling framework can correctly:

- walk the mesh using the shared-vertex connectivity for geometry operations (stream tracing, surface intersection, etc.);
- read solution values element-locally, preserving the jumps at shared interfaces.

For element-local DOFs at non-vertex positions (interior quadrature points, face quadrature points, etc.), use `GridLocation_t = InterpolationPoints` with `SolutionInterpolation_t` metadata from CPEX 0045. `GridLocation_t = Vertex` with a `PointList` is specifically the correct choice when the DOF positions coincide with mesh vertices and the shared-vertex mesh connectivity is to be preserved.

`GridLocation_t = Vertex` with `SolutionDiscontinuous` and *no* `PointList` is not valid: the SIDS `DataSize` formula for `Vertex` yields `DataSize` = `VertexSize` (the shared-vertex count), which cannot represent the larger element-local DOF count. Similarly, `PointRange` is not suitable for this case because a contiguous range of vertex indices cannot express the element-local repetition of shared vertices; only `PointList` provides the necessary per-element DOF ordering. `GridLocation_t = CellCenter` with `SolutionDiscontinuous` remains valid since cell-centered data is inherently one value per element with no shared-entity ambiguity.

This approach extends naturally to mixed-element meshes (`MIXED`, `NGON_n`, `NFACE_n`). The `PointList` simply lists vertex indices for each element in element order; elements with different vertex counts contribute different numbers of entries. The `SolutionContinuity_t = SolutionDiscontinuous` flag tells readers that repeated vertex indices represent independent DOFs regardless of element type heterogeneity.

SIDS Amendment (PointList semantics under Discontinuous solutions): When `SolutionContinuity_t = SolutionDiscontinuous` and a `PointList` is present under `FlowSolution_t` with `GridLocation_t = Vertex`, repeated vertex indices are permitted and represent element-local DOFs. The DOF-to-element mapping is defined by traversal of the zone's `Elements_t` sections in ascending section index order, then by each section's connectivity

array order. Solution array position i corresponds to the i-th entry of the `PointList` under this traversal. This is an explicit amendment to the `PointList` description in SIDS §7.7 for this case; repeated indices under a continuous solution remain undefined and invalid.

Note: For element-local DOFs at non-vertex positions, or when the implicit element-ordering covenant above is inconvenient, `GridLocation_t = InterpolationPoints` with `SolutionInterpolation_t` from CPEX 0045 is the preferred encoding and does not require repeated indices.

See Example 6.7 for the complete C encoding of this pattern.

3.9 Interaction with `Rind_t` (Ghost Layers)

`SolutionContinuity_t` applies uniformly to the entire solution field, including any rind (ghost) entries declared by `Rind_t`. A `SolutionDiscontinuous` label means that all data entries—interior and rind alike—are element-local: each rind entry belongs to one element and must not be interpreted as a shared vertex value. DG solvers that use ghost layers for MPI halo exchange may therefore store element-local rind DOFs in the standard way; the `SolutionDiscontinuous` label covers them. Rind entries of a `SolutionContinuous` solution retain their usual shared-DOF semantics. No special treatment of the rind is required beyond what `Rind_t` already specifies; the continuity label is consistent across the full data array.

4 Interaction with CPEX 0045 (Higher-Order Elements)

CPEX 0045 introduced `GridLocation_t = InterpolationPoints`, `SolutionInterpolation_t`, and the concept of per-element-type polynomial basis definitions. The present CPEX complements rather than modifies that infrastructure:

- CPEX 0045 answers: "What polynomial basis and order describes the solution within each element?"
- This CPEX answers: "Are the resulting DOFs shared across element interfaces or element-local?"

Both pieces of metadata are needed for a complete description of a higher-order solution field. They are orthogonal:

	Continuous	**Discontinuous**
Lagrange $p = 1$	CG FEM	DG $p = 1$
Lagrange $p = 3$	CG spectral	DG spectral
Hierarchical $p = 2$	CG (modal)	DG (modal)
$p = 0$ (constant)	—	FV / DG $p = 0$

For higher-order solutions using `GridLocation_t = InterpolationPoints`, the recommended practice is:

1. Use CPEX 0045 to define the element type, interpolation basis, and polynomial order via `SolutionInterpolation_t`.
2. Use this CPEX to set `SolutionContinuity_t` on the `FlowSolution_t` node to indicate whether the field is continuous or discontinuous.

For lower-order solutions using `GridLocation_t = Vertex` or `CellCenter`, this CPEX can be used independently of CPEX 0045. See Section 3.8 for the prescribed data layout when DG DOFs coincide with shared mesh vertices.

5 Backward Compatibility

- **Existing files:** Fully compatible. The `SolutionContinuity` node is optional; its absence is equivalent to `SolutionContinuityNull`. No existing file is invalidated.
- **Existing readers:** Unaffected. Per the CGNS convention, readers silently ignore unknown child nodes. An older library reading a file with `SolutionContinuity` nodes will simply skip them.
- **Existing writers:** Unaffected. Codes that do not call the new API functions produce files without `SolutionContinuity` nodes, identical to the current behavior.
- **ABI:** The `cgns_sol`, `cgns_discrete`, and `cgns_subreg` structures each grow by two fields (one `int`-sized enum and one `double` node identifier). This is an ABI change requiring recompilation of code that directly accesses the internal structures. The public API is purely additive (new functions, new enum), so no existing function signatures change.
- **Parallel I/O (`cgp_*`):** The `SolutionContinuity` metadata node is a small scalar child of `FlowSolution_t` and carries no per-element data. It is written and read via the standard collective-write / independent-read model, identical to how `GridLocation` is handled in the parallel MLL. No new `cgp_*` variants are required.

6 Examples

6.1 Example 1: Writing a Continuous Galerkin Solution (C)

```c
/* Create a vertex-located solution node */
int S;
cg_sol_write(fn, B, Z, "FlowSolution", CGNS_ENUMV(Vertex), &S);

/* Mark as continuous (CG FEM) */
cg_sol_continuity_write(fn, B, Z, S,
    CGNS_ENUMV(SolutionContinuous));

/* Write solution data (one value per mesh vertex, shared) */
cg_field_write(fn, B, Z, S, CGNS_ENUMV(RealDouble),
    "Pressure", pressure, &F);
```

6.2 Example 2: Writing a Discontinuous Galerkin Solution (C)

```c
/* Create an interpolation-point-located solution node */
int S;
cg_sol_write(fn, B, Z, "FlowSolution_DG",
    CGNS_ENUMV(InterpolationPoints), &S);

/* Set polynomial order (CPEX 0045) */
cg_sol_interpolation_order_write(fn, B, Z, S, 3, 0);

/* Mark as discontinuous */
cg_sol_continuity_write(fn, B, Z, S,
```

```
    CGNS_ENUMV(SolutionDiscontinuous));

/* Write element-local DOFs */
cg_field_write(fn, B, Z, S, CGNS_ENUMV(RealDouble),
    "Pressure", pressure_dg, &F);
```

6.3 Example 3: Reading and Dispatching on Continuity (C)

```
CGNS_ENUMT(SolutionContinuity_t) repr;
cg_sol_continuity_read(fn, B, Z, S, &repr);

switch (repr) {
case CGNS_ENUMV(SolutionContinuous):
    /* Shared DOFs: interpolate using global mesh connectivity */
    visualize_continuous(fn, B, Z, S);
    break;
case CGNS_ENUMV(SolutionDiscontinuous):
    /* Element-local DOFs: render per-element, preserve jumps */
    visualize_discontinuous(fn, B, Z, S);
    break;
default:
    /* Null or UserDefined: fall back to legacy heuristics */
    visualize_legacy(fn, B, Z, S);
    break;
}
```

6.4 Example 4: HDG Solver with Both Continuity Types (C)

```
/* Volume DOFs (discontinuous) */
int S_vol;
cg_sol_write(fn, B, Z, "VolumeSolution",
    CGNS_ENUMV(InterpolationPoints), &S_vol);
cg_sol_continuity_write(fn, B, Z, S_vol,
    CGNS_ENUMV(SolutionDiscontinuous));

/* Trace DOFs on element faces (continuous on skeleton).
 * Case A: p=0 traces (one DOF per face) -- use FaceCenter. */
int S_trace;
cg_sol_write(fn, B, Z, "TraceSolution",
    CGNS_ENUMV(FaceCenter), &S_trace);
cg_sol_continuity_write(fn, B, Z, S_trace,
    CGNS_ENUMV(SolutionContinuous));

/* Case B: higher-order traces -- use InterpolationPoints
 * with CPEX 0045 metadata for the face polynomial basis. */
int S_trace_ho;
cg_sol_write(fn, B, Z, "TraceSolution_HO",
    CGNS_ENUMV(InterpolationPoints), &S_trace_ho);
cg_sol_interpolation_order_write(fn, B, Z, S_trace_ho, 2, 0);
cg_sol_continuity_write(fn, B, Z, S_trace_ho,
    CGNS_ENUMV(SolutionContinuous));
```

6.5 Example 5: Fortran Usage

```
integer :: S, ier, repr

! Write a continuous solution
call cg_sol_write_f(fn, B, Z, 'FlowSolution', Vertex, S, ier)
call cg_sol_continuity_write_f(fn, B, Z, S, &
    SolutionContinuous, ier)

! Read continuity
call cg_sol_continuity_read_f(fn, B, Z, S, repr, ier)
if (repr == SolutionDiscontinuous) then
    ! Element-local interpolation
end if
```

6.6 Example 6: Low-Order Finite Volume (DG $p = 0$)

```
/* Cell-centered FV solution */
int S;
cg_sol_write(fn, B, Z, "FlowSolution",
    CGNS_ENUMV(CellCenter), &S);

/* Explicitly mark as discontinuous */
cg_sol_continuity_write(fn, B, Z, S,
    CGNS_ENUMV(SolutionDiscontinuous));
```

This example illustrates that continuity metadata provides value even for traditional low-order methods. Although a `CellCenter` solution is inherently element-local, explicitly recording the continuity assists post-processors and coupling frameworks in programmatically distinguishing between solution types.

6.7 Example 7: DG Solution on a Shared-Vertex Mesh (C)

This example addresses the case where the mesh uses shared vertices (enabling stream tracing and mesh walking via standard `Elements_t` connectivity), but each element stores its own independent solution values at those vertices. The `SolutionContinuity_t` node is the flag that tells readers the repeated vertex indices in the `PointList` are independent DOFs, not duplicates.

```
/*
 * Mesh: nVerts shared vertices, nElems elements,
 *       nVertsPerElem vertices per element.
 * Solution: element-local DOFs at each element's vertices.
 *           Total DOFs = nElems * nVertsPerElem.
 */

/* Build the point list: for each element, list its vertex indices.
 * Shared vertices appear multiple times -- once per adjacent element. */
cgsize_t point_list[nElems * nVertsPerElem];
for (int e = 0; e < nElems; e++)
    for (int v = 0; v < nVertsPerElem; v++)
        point_list[e * nVertsPerElem + v] =
            element_connectivity[e][v];  /* 1-based vertex index */

/* Create solution node with PointList using the idiomatic
 * cg_sol_ptset_write: Vertex location preserves shared-mesh
 * connectivity for geometry operations.
 * DataSize = nElems * nVertsPerElem (not nVerts). */
int S;
cg_sol_ptset_write(fn, B, Z, "FlowSolution_DG",
    CGNS_ENUMV(Vertex),
    CGNS_ENUMV(PointList),
```

```
    (cgsize_t)(nElems * nVertsPerElem),
    point_list, &S);

/* Mark as discontinuous: values at repeated vertex indices
 * are independent across elements, ordered by Elements_t
 * section index then connectivity order. */
cg_sol_continuity_write(fn, B, Z, S,
    CGNS_ENUMV(SolutionDiscontinuous));

/* Write element-local solution data */
cg_field_write(fn, B, Z, S, CGNS_ENUMV(RealDouble),
    "Pressure", pressure_dg, &F);
```

A reader distinguishes this from a continuous solution as follows:

```
CGNS_ENUMT(SolutionContinuity_t) cont;
cg_sol_continuity_read(fn, B, Z, S, &cont);

if (cont == CGNS_ENUMV(SolutionDiscontinuous)) {
    /* PointList entries are element-local DOF indices.
     * Values at the same vertex index from different elements
     * are independent -- do not average. */
}
```

6.8 Example 8: DiscreteData_t with Discontinuous Error Indicator (C)

This example shows a DG error indicator stored in a `DiscreteData_t` node. The usage mirrors `FlowSolution_t`; the only difference is the API prefix (`cg_discrete_` instead of `cg_sol_`).

```
/* Create a cell-centered discrete data node for error indicators */
int D;
cg_discrete_write(fn, B, Z, "ErrorIndicator", &D);
cg_goto(fn, B, "Zone_t", Z, "DiscreteData_t", D, "end");
cg_gridlocation_write(CGNS_ENUMV(CellCenter));

/* Mark as discontinuous (element-local error estimates) */
cg_discrete_continuity_write(fn, B, Z, D,
    CGNS_ENUMV(SolutionDiscontinuous));

/* Write per-element error data */
cg_goto(fn, B, "Zone_t", Z, "DiscreteData_t", D, "end");
cg_array_write("ElementError", CGNS_ENUMV(RealDouble),
    1, &nElems, error_data);
```

6.9 Example 9: ZoneSubRegion_t with Boundary Flux Data (C)

This example stores discontinuous boundary flux data on a `ZoneSubRegion_t` that references a boundary condition region.

```
/* Create a sub-region referencing a BC */
int SR;
/* CellDimension is the zone's cell dimension (e.g., 3 for a 3-D volume
 * zone), so CellDimension - 1 is the face dimension. */
cg_subreg_bcname_write(fn, B, Z, "WallFluxes",
    CellDimension - 1, "Wall_BC", &SR);

/* Mark as discontinuous (each face owns its own flux values) */
cg_subreg_continuity_write(fn, B, Z, SR,
    CGNS_ENUMV(SolutionDiscontinuous));

```

```c
/* Write flux data on the sub-region */
cg_goto(fn, B, "Zone_t", Z, "ZoneSubRegion_t", SR, "end");
cg_array_write("HeatFlux", CGNS_ENUMV(RealDouble),
    1, &nBCFaces, heat_flux);
```

6.10 Example 10: Solver Restart with Continuity-Aware Read (C)

A solver reading a restart file must reconstruct the solution state correctly based on the continuity. This example shows how a restart reader uses the continuity metadata to dispatch between shared-DOF and element-local reconstruction.

```c
/* Read solution metadata */
char sol_name[CGIO_MAX_NAME_LENGTH+1];
CGNS_ENUMT(GridLocation_t) loc;
cg_sol_info(fn, B, Z, S, sol_name, &loc);

CGNS_ENUMT(SolutionContinuity_t) cont;
cg_sol_continuity_read(fn, B, Z, S, &cont);

/* Read the field data.  In a generic restart loop, call
 * cg_field_info(fn,B,Z,S,F,...) first to discover field names
 * and types before calling cg_field_read. */
cg_field_read(fn, B, Z, S, "Pressure",
    CGNS_ENUMV(RealDouble), range_min, range_max,
    pressure_buf);

/* Reconstruct based on continuity */
if (cont == CGNS_ENUMV(SolutionContinuous)) {
    /* DOFs are shared at mesh vertices/edges/faces.
     * Map directly to the global DOF vector. */
    reconstruct_continuous(pressure_buf, loc, mesh);
} else if (cont == CGNS_ENUMV(SolutionDiscontinuous)) {
    /* DOFs are element-local.  Each element owns its
     * own copy of values at shared interfaces. */
    reconstruct_discontinuous(pressure_buf, loc, mesh);
} else {
    /* Null: legacy file.  Infer from data size or
     * assume continuous for backward compatibility. */
    reconstruct_legacy(pressure_buf, loc, mesh);
}
```

6.11 Example 11: FSI Coupling — Structural and Fluid Solutions (C)

In Fluid-Structure Interaction (FSI) applications, a single zone may carry both fluid and structural solution fields with different continuity types. A continuous FEM structural solver stores shared displacement DOFs, while a DG fluid solver stores element-local flow variables. Separate `FlowSolution_t` nodes with distinct continuity metadata allow coupling frameworks to correctly interpret and transfer each field.

```c
/* Structural solution: continuous FEM displacements */
int S_struct;
cg_sol_write(fn, B, Z, "StructuralDisplacement",
    CGNS_ENUMV(Vertex), &S_struct);
cg_sol_continuity_write(fn, B, Z, S_struct,
    CGNS_ENUMV(SolutionContinuous));
cg_field_write(fn, B, Z, S_struct, CGNS_ENUMV(RealDouble),
    "DisplacementX", disp_x, &F);
cg_field_write(fn, B, Z, S_struct, CGNS_ENUMV(RealDouble),
    "DisplacementY", disp_y, &F);

```

```c
/* Fluid solution: discontinuous DG flow field */
int S_fluid;
cg_sol_write(fn, B, Z, "FluidSolution",
    CGNS_ENUMV(InterpolationPoints), &S_fluid);
cg_sol_interpolation_order_write(fn, B, Z, S_fluid, 3, 0);
cg_sol_continuity_write(fn, B, Z, S_fluid,
    CGNS_ENUMV(SolutionDiscontinuous));
cg_field_write(fn, B, Z, S_fluid, CGNS_ENUMV(RealDouble),
    "Pressure", pressure, &F);
cg_field_write(fn, B, Z, S_fluid, CGNS_ENUMV(RealDouble),
    "VelocityX", vel_x, &F);

/* A coupling framework reading this zone can query
 * each FlowSolution_t's continuity to determine the
 * correct interpolation and transfer strategy:
 *   - Continuous fields: global shared-DOF interpolation
 *   - Discontinuous fields: element-local projection */
```

The same pattern applies when the structural and fluid domains are in separate zones. The continuity metadata on each zone's `FlowSolution_t` nodes enables the coupling framework to apply the correct inter-code transfer operators without relying on naming conventions or external configuration.

6.12 Example 12: Conjugate Heat Transfer — Perfect and Imperfect Thermal Contact (C)

Conjugate heat transfer (CHT) simulations couple a fluid temperature field (often stored at cell centers or DG DOFs) with a solid temperature field (often stored as a continuous CG FEM solution). The continuity metadata disambiguates the two layouts and informs the coupling framework how to perform the interface transfer.

```c
/* ---- Fluid zone: FV/DG temperature (discontinuous) ---- */
int S_fluid;
cg_sol_write(fn, B, fluidZone, "ThermalSolution",
    CGNS_ENUMV(CellCenter), &S_fluid);
cg_sol_continuity_write(fn, B, fluidZone, S_fluid,
    CGNS_ENUMV(SolutionDiscontinuous));
cg_field_write(fn, B, fluidZone, S_fluid,
    CGNS_ENUMV(RealDouble), "Temperature", T_fluid, &F);

/* ---- Solid zone: CG FEM temperature (continuous) ---- */
int S_solid;
cg_sol_write(fn, B, solidZone, "ThermalSolution",
    CGNS_ENUMV(Vertex), &S_solid);
cg_sol_continuity_write(fn, B, solidZone, S_solid,
    CGNS_ENUMV(SolutionContinuous));
cg_field_write(fn, B, solidZone, S_solid,
    CGNS_ENUMV(RealDouble), "Temperature", T_solid, &F);

/* ---- Interface heat flux on the solid side (face-local) ---- */
int SR;
cg_subreg_bcname_write(fn, B, solidZone, "FluidInterface",
    CellDimension - 1, "FluidInterface_BC", &SR);
cg_subreg_continuity_write(fn, B, solidZone, SR,
    CGNS_ENUMV(SolutionDiscontinuous));
cg_goto(fn, B, "Zone_t", solidZone,
    "ZoneSubRegion_t", SR, "end");
cg_array_write("HeatFlux", CGNS_ENUMV(RealDouble),
    1, &nInterfaceFaces, heat_flux);
```

With *perfect* thermal contact the fluid-side and solid-side interface temperatures are equal; the coupling framework can enforce this constraint knowing the solid temperature is `SolutionContinuous`

(shared vertex DOFs) and the fluid temperature is `SolutionDiscontinuous` (cell-centered).

With *imperfect* thermal contact (finite thermal resistance), a temperature jump ΔT exists at the interface. Each domain retains its own interface temperature, and the coupling framework applies the resistance relation $q = \Delta T / R$ between them. No change to the `SolutionContinuity_t` labels is needed: the fluid and solid temperatures are stored and tagged independently in their respective zones, and the jump is an emergent result of the coupling rather than a property of either field's storage layout.

7 SIDS Impact

1. **New enumeration type:** `SolutionContinuity_t` is added to the SIDS building-block structure definitions (Section 4 of the SIDS document), alongside existing enumeration types such as `GridLocation_t` (Section 4.5) and `DataClass_t` (Section 4.1).
2. **FlowSolution_t amendment:** The `FlowSolution_t` node definition (SIDS Section 7.7) is amended to include an optional `SolutionContinuity_t` child node.
3. **DiscreteData_t amendment:** The `DiscreteData_t` node definition (SIDS Section 12.4) is similarly amended.
4. **ZoneSubRegion_t amendment:** The `ZoneSubRegion_t` node definition (SIDS Section 7.9) is similarly amended.
5. **PointList semantics amendment:** The description of `PointList` under `FlowSolution_t` in SIDS §7.7 is amended to permit repeated vertex indices when `SolutionContinuity_t = SolutionDiscontinuous`. Repeated indices in that context represent element-local DOFs and are not malformed duplicates; the DOF-to-element mapping follows ascending `Elements_t` section index order, then connectivity order within each section. This amendment does not affect `PointList` semantics under any other condition.
6. No existing SIDS node types, enumerations, or conventions are modified or removed.

8 File Mapping Impact

The `SolutionContinuity` child node is stored identically to `GridLocation`: as a character node containing the string name of the enumeration value. The file-mapping attributes are:

Attribute	Value
Node name	`SolutionContinuity`
Label	`SolutionContinuity_t`
Data type	`C1` (character)
Dimension	1
Dimension value	length of enumeration string
Data	enumeration name string

This follows the same pattern as the `GridLocation` node under `FlowSolution_t`: node name `"GridLocation"`, label `"GridLocation_t"`, data type `"C1"`, with the enumeration string as data.

The node is written only when the continuity is not `SolutionContinuityNull`. When absent, readers default to `SolutionContinuityNull`, paralleling how an absent `GridLocation` node defaults to `Vertex`.

9 Design Decisions

Several design questions were evaluated during the development of this proposal. Each is addressed below with rationale grounded in existing CGNS conventions and SIDS precedent.

1. **Scope: `BCData_t` is excluded.** `SolutionContinuity_t` is *not* added to `BCData_t`. Boundary condition data arrays in `BCData_t` are sized by the associated `PointList`/`PointRange` and `GridLocation_t`, which already determine the data layout unambiguously. In DG methods, boundary conditions are typically applied weakly through numerical fluxes; the element-local nature of the boundary data follows from the solution continuity on the parent `FlowSolution_t` node. Adding `SolutionContinuity_t` to `BCData_t` would be redundant and would introduce a potential inconsistency if the BC continuity differs from the solution it constrains. Should a future use case require per-BC continuity metadata (e.g., coupling frameworks imposing boundary data in a different continuity than the interior, or interface conditions in CHT simulations with imperfect thermal contact where a flux-jump boundary condition is prescribed), it can be proposed as a separate, narrowly scoped CPEX without affecting this proposal. The `ZoneSubRegion_t` extension introduced here partially addresses such cases: interface fluxes or surface-local fields that require a continuity label can be stored in a `ZoneSubRegion_t` referencing the relevant boundary condition (see Example 6.12).

2. **Per-solution-node granularity (not per-field).** The continuity applies at the `FlowSolution_t` level, meaning all `DataArray_t` children share the same continuity. This follows the precedent set by `GridLocation_t`, which is likewise a per-solution-node property: all fields within a `FlowSolution_t` node share the same grid location. Per-`DataArray_t` granularity was rejected because it would add significant API and metadata complexity with no demonstrated need. Solvers that genuinely mix continuity types within a single zone (e.g., HDG methods with continuous traces and discontinuous volumes) should use separate `FlowSolution_t` nodes, exactly as they would use separate nodes for fields at different `GridLocation_t` values.

3. **`CellCenter` + `SolutionContinuous`: permitted but semantically inconsistent.** The MLL does *not* enforce consistency between `SolutionContinuity_t` and `GridLocation_t`. A `CellCenter` solution is inherently element-local, making `SolutionContinuous` semantically contradictory, but the library shall not reject such combinations. This follows the established CGNS convention of being permissive in what is accepted: the standard documents correct usage; validation is delegated to application-level tools such as `cgnscheck`.

   Convention: `SolutionContinuity_t = SolutionContinuous` with `GridLocation_t = CellCenter` should not be used. `cgnscheck` should issue a warning when this combination is encountered.

4. **Recommended `cgnscheck` validation rules.** The following consistency checks are recommended for `cgnscheck` and similar validation tools:

   Warning: `SolutionContinuity_t = SolutionContinuous` with `GridLocation_t = CellCenter` — semantically inconsistent since cell-centered data is inherently element-

local (Design Decision 3 above).

Error: `SolutionContinuity_t = SolutionDiscontinuous` with `GridLocation_t = Vertex` and *no* `PointList` — the SIDS `DataSize` formula yields `VertexSize` (the shared-vertex count), which cannot represent the larger element-local DOF count (Section 3.8).

Warning: `SolutionContinuity_t` value outside the range $[0, \texttt{NofValidSolutionContinuity})$ — invalid enumeration value.

10 Summary of API Additions

Symbol	Description
`SolutionContinuity_t`	New enumeration type: `Null`, `UserDefined`, `SolutionContinuous`, `SolutionDiscontinuous`.
`SolutionContinuityName`	String table for the enumeration values.
`NofValidSolutionContinuity`	Count of valid enumeration values (4).
`cg_SolutionContinuityName`	C: return the string name for a `SolutionContinuity_t` value (mirrors `cg_GridLocationName`).
`cg_sol_continuity_write`	C: write solution continuity for a `FlowSolution_t` node.
`cg_sol_continuity_read`	C: read solution continuity for a `FlowSolution_t` node.
`cg_discrete_continuity_write`	C: write solution continuity for a `DiscreteData_t` node.
`cg_discrete_continuity_read`	C: read solution continuity for a `DiscreteData_t` node.
`cg_subreg_continuity_write`	C: write solution continuity for a `ZoneSubRegion_t` node.
`cg_subreg_continuity_read`	C: read solution continuity for a `ZoneSubRegion_t` node.
`cg_sol_continuity_write_f`	Fortran: write solution continuity (`FlowSolution_t`).
`cg_sol_continuity_read_f`	Fortran: read solution continuity (`FlowSolution_t`).
`cg_discrete_continuity_write_f`	Fortran: write solution continuity (`DiscreteData_t`).
`cg_discrete_continuity_read_f`	Fortran: read solution continuity (`DiscreteData_t`).
`cg_subreg_continuity_write_f`	Fortran: write solution continuity (`ZoneSubRegion_t`).
`cg_subreg_continuity_read_f`	Fortran: read solution continuity (`ZoneSubRegion_t`).

11 References

1. CGNS Standard Interface Data Structures (SIDS),
   https://cgns.org/standard/SIDS/CGNS_SIDS.html
2. CGNS Mid-Level Library—Flow Solution,
   https://cgns.org/standard/MLL/api/c_api.html
3. CPEX 0045, "Polynomial Data and Curved Grid Elements,"
   https://github.com/CGNS/CGNS/issues/577
4. Hesthaven, J.S. and Warburton, T., *Nodal Discontinuous Galerkin Methods*, Springer, 2008.
5. Cockburn, B., Karniadakis, G.E., and Shu, C.-W., eds., *Discontinuous Galerkin Methods: Theory, Computation, and Applications*, Springer, 2000.