Unstructured Hybrid Mesh Support for Overture
A Description of the Ugen and AdvancingFront Classes and Documentation for Additional Support Classes

Kyle K. Chand
Centre for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, 94551
chand@llnl.gov
http://www.llnl.gov/casc/people/chand
http://www.llnl.gov/casc/Overture

May 20, 2011

Abstract

Overture's support for unstructured and hybrid mesh generation is implemented through the classes described in this document. There are three tiers of classes used for the generation of unstructured meshes. The first tier consists of container classes, essentially `GeometricADT`, used by the mesh generator to perform geometric searches. The `AdvancingFront` class encapsulates the logic of an advancing front mesh generator and uses `GeometricADT`'s. Finally, `Ugen` acts as the interface between the unstructured mesh generator and the rest of Overture.

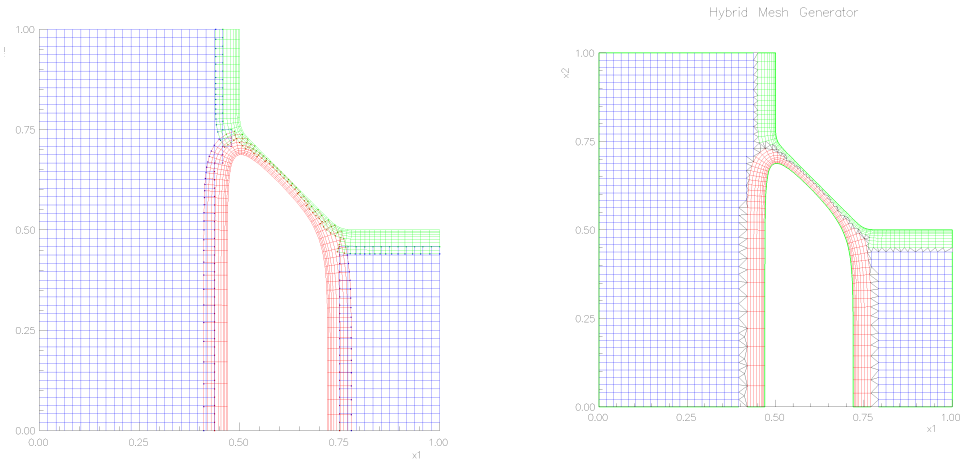
The following classes are described in this document:

- Ugen : Unstructured/Hybrid mesh generator interface
- AdvancingFront : unstructured mesh generation using the advancing front method
- GeometricADT : bounding box alternating digital tree for geometric searches
- GeometricADT::iterator : defines an iteration path through a GeometricADT
- GeometricADT::traversor : performs a search traversal of a GeometricADT
- NTreeNode<int degree, class Data> : primitive container class used to build GeometricADT
- Exceptions : a list of the various exceptions thrown by the above classes
- CompositeGridHybridConnectivity : manages the mappings between unstructured and structrutred components of a CompositeGrid

Contents

1 Introduction

Hybrid meshes consist of regions of structured grids joined by unstructured meshes. Figure 1 compares overlapping and hybrid meshes for the same geometry. Generation of unstructured hybrid meshes in Overture is orchestrated by the class `Ugen`. When provided with a `CompositeGrid`, `Ugen` removes the overlap, determines the hole boundaries and conducts the generation of an unstructured mesh filling the spaces between component grids. Currently, the `CompositeGrid` must contain an overlapping grid with holes cut using the ``‘‘compute for hybrid mesh’’`` option in `Ogen`. To generate unstructured meshes, `Ugen` utilizes an advancing front method implemented in the class `AdvancingFront`. One the unstructured regions have been generated they are placed into an `UnstructuredMapping` and subsequently added to the original CompositeGrid. Connectivity between the structured and unstructured regions can be accessed through the class `CompositeGridHybridConnectivity`, an instance of which is created in the original `CompositeGrid`.



(a) (b)

Figure 1: Comparison of Overlapping and Hybrid Meshes : (a) Overlapping Grid; (b) Hybrid Mesh

2 Generating Hybrid Meshes

The environment provided by `Ugen` assumes that mappings have already been created and that the user is prepared to assemble a hybrid mesh. Selecting the “generate hybrid mesh” option in the main `Ogen` menu initially proceeds in a manner similar to the overlapping grid version. A list of the mappings involved in the construction of the hybrid mesh is built. As with the overlapping grid case, a priority is assigned to each mapping with selections being made

in ascending order of priority. Higher priority grids will cut holes in lower priority ones. Figure **??** displays the initial state of the hybrid mesh generation environment just after the mappings for Figure **??**a have been selected. There are several options now available via the pop-up menu :

- set plotting frequency (¡1 for never) : This options selects how many unstructured elements will be generated before the plot is updated. Setting the value to -1 forces the mesh generator to continue until the mesh is completely generated or an error occurs.
- continue generation : continue generating the unstructured mesh until the plotting frequency is reached; the mesh is complete; or an error occurs.
- enlarge hole : enlarge the hole and reinitialize the mesh. This option will destroy an already generated unstructured mesh
- reset hole : destroy any unstuructured mesh elements and reinitialize the algorithm
- plot component grids (toggle) : plot/do not plot component grids
- plot control function (toggle) : plot/do not plot the control function
- open graphics : open a graphics dump file
- plot object : plot/refresh the image
- change the plot : change plotting characteristics such as the component grids and control function
- exit : exit the hybrid mesh generation interface, this will attempt to add the unstructured mesh to the `CompositeGrid` and generate the connectivity between the unstructured and structured meshes.

2.1 Preprocessing the Composite Grids

2.2 Class Ugen

2.3 Default Constructor

Ugen()

2.4 Constructor

Ugen(PlotStuff & ps_)

ps_ (input) : reference to the `PlotStuff` that the `Ugen` should use for plots

2.5 updateHybrid

void
updateHybrid(CompositeGrid & cg,
MappingInformation & mapInfo)

cg (input) : `CompositeGrid` on which to construct a hybrid mesh

mapInfo (input) : currently only provides a `PlotStuff` instance for plotting

Description : `updateHybrid` performs the following tasks, primarily through private and protected methods

- strips away the overlap in `cg`
- determines the faces comprising the initial holes
- initializes an `AdvancingFront` with the faces determined above
- optionally generates or destroys mesh stretching influences on the `AdvancingFront`
- provides the `PlotStuff`/command line interface for plotting and interactive mesh generation

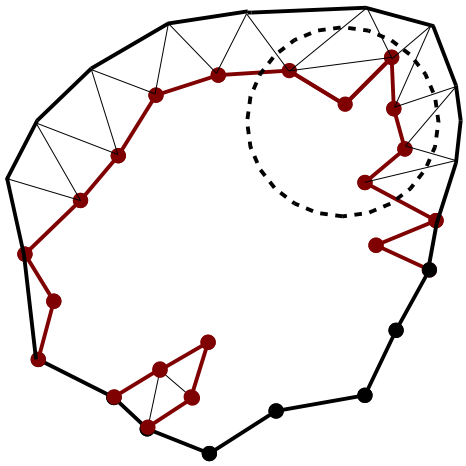


Figure 2: An advancing front (red) grows off of an initial curve (black)

Currently, this method catches all exceptions arising from failures in the `AdvancingFront`'s mesh generation algorithms. Low level data structure errors are, in general, not caught and are thrown through to the calling scope. It is assumed that if a low level error occurs (say in a `GeometricADT` or, worse, an `NTreeNode`, that the state is corrupt enough that `updateHybrid` cannot recover. Generally, `updateHybrid` will recover from a mesh generation error by plotting the current state of the hybrid mesh and allowing a limited set of `Ugen` manipulations.

2.6 computeZoneMasks

void
computeZoneMasks(CompositeGrid &cg, intArray * &zoneMasks, IntegerArray &numberOfMaskedZones)

cg (input/output) : the composite grid containing the mapped grids requiring zone masks, vertex masks may be adjusted

zoneMasks (output) : an array of IntegerArray's for each grid containing the mask for each zone (¡1 means a zone is masked out)

numberOfMaskedZones : an IntegerArray of the number of zones in each grid in cg that were masked out

Description : a zone will be considered masked out if any one of its vertices have `MappedGrid::mask` value ¡=0 or if all of the surrounding zone meet this criteria. One layer of ghost zones is included.
WARNING :: if a zone is found to be floating in a sea of masked zones (ie it is isolated) then it and its constituent vertices are masked out as well; this adjusts the mapping's vertex mask array.

3 Advancing Front Mesh Generation

Advancing front mesh generation creates an unstructured mesh by "growing" elements off of an initial set of curves (2D) or surfaces (3D) which describe a connected domain. Discretized representations of these curves or surfaces are collectively known as the "front" which is successively "advanced" until the domain is filled with an unstructured mesh. Figure 2 illustrates an advancing front. In this case, the initial curve is bold black, the front is bold red with dots and the generated elements are black The implmentation of the mesh generator follows that of Peraire, J, Peiro, J. and Morgan, K., Ref ??.

3.0.1 Mesh Generation Algorithm

Advancement of the "advancing" front refers the the process by which new elements are created. The process may be summarized as :

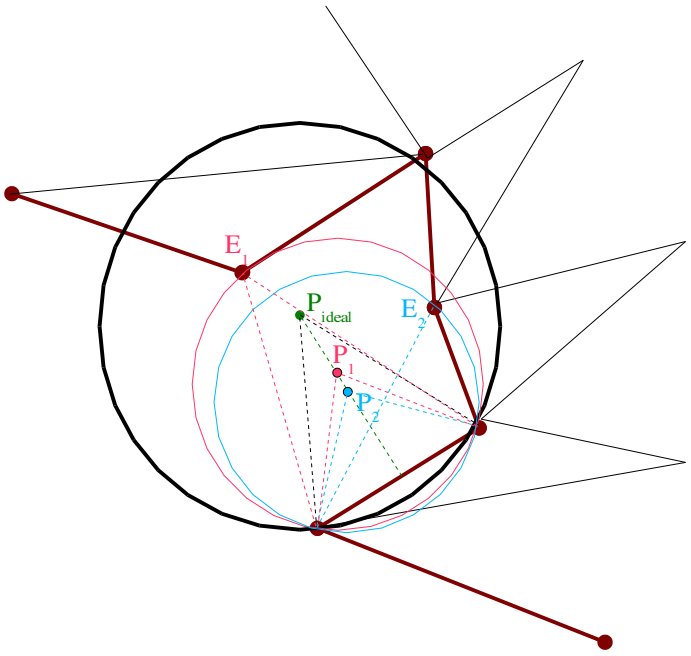


Figure 3: A cluttered view of candidate vertices

1. Choose and discretize an initial boundary
2. Select a face to advance
3. Find all nearby old vertices
4. Calculate possible new vertices
5. Prioritize and choose a vertex from the list of old and new vertices
6. Delete the starting face from the front and add any new faces
7. Return to 2 and repeat until the front is empty.

Initially, a discretized curve (2D) or surface (3D) must be created that encloses the region to be meshed. Assuming we start with 2D curve discretized into a set of line segments (called "faces"), mesh generation consists of generating elements/zones from faces in the front. As faces are used they are deleted from the front, while new faces are created and added to it. Eventually the front will collapse upon itself, becoming empty, resulting in a completed mesh. Zooming into the dashed circle of Figure 2 provides an opportunity to examine a sample front advance. Figure 3 shows a face chosen for advancement and marks several important features of the algorithm. The letter symbols indicate the points that could be used to complete a new element and dashed lines represent the new elements they would create. A new element may be created by using existing points belonging to other faces in the front or a new point may be created.

A user defined notion of the "ideal" element in that region of the mesh leads to the calculation of the "ideal" new point, P_{ideal}. In a uniform 2D mesh the ideal element would be an equliateral triangle, but typically mesh size and quality control parameters can permit stretched elements. Candidate points that already exist in the front are sought within a circle with an origin at P_{ideal} and passing throught the points in the face. Figure 3 shows this search circle in bold black and marks the candidates already in the front as E_1 and E_2. An element can also be created by generating a new point. In addition to the "ideal" point, the locations of new points are the centers of the circles created using the two edge points and each existing candidate front point. In this case, the two circles are shown in red and blue for the circles using E_1 and E_2 respectively. P_1 and P_2 are the additional new points to be considered. Now the algorithm must choose which point to use, or, equivalently, which element to create.

The candidates $E_1, E_2, P_{\text{ideal}}, P_1, P_2$, are prioritized and assembled into a queue. Vertices that already exist, E_1 and E_2 are ordered according to how close the element they would create is to the "ideal" element. Next comes the actual "ideal" vertex P_{ideal}. Then the additional candidate new (P_1, P_2) vertices are ordered in a similar way to

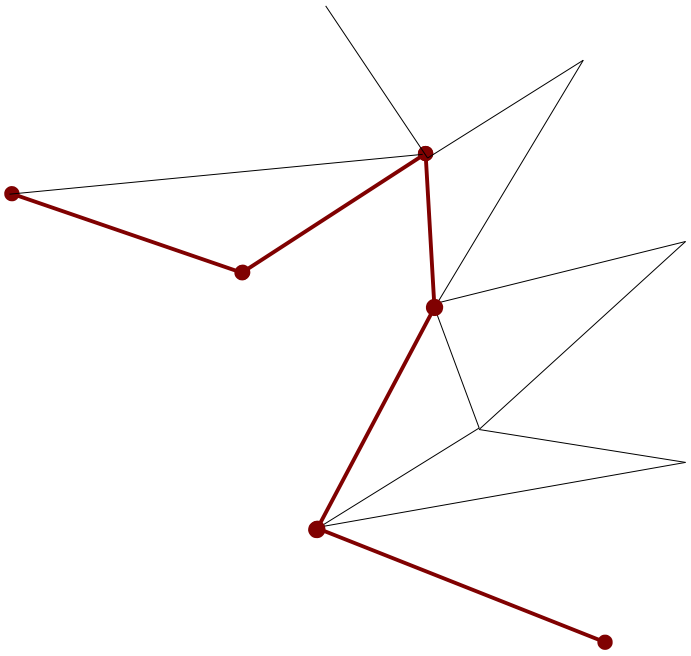


Figure 4: The new front

the existing vertices, the closer to "ideal" the element would be, the higher the priority. In Figure 3, the ordering would be : $E_1, E_2, P_{\text{ideal}}, P_1, P_2$. The first candidate in this queue that generates a valid element (no intersections) is chosen to complete the new element. Once the new element has been created any new faces are added to the front. The original face, now buried beneath the front, is removed. Figure 4 illustrates the end result of our example. This process is repeated until the front is empty.

3.0.2 Mesh stretching control

you'll have to wait for this subsection

3.1 Class AdvancingFront

3.2 Default Constructor

AdvancingFront()

Purpose: Create an uninitialized AdvancingFront.

3.3 Constructor

AdvancingFront(intArray &initialFaces,
realArray &xyz_in,
Mapping *backgroundMapping_ = NULL)

Purpose: Create an AdvancingFront initialized with a set of initial faces and a background mapping.

initialFaces (input) : the list of initial faces (vertex lists for each face)

xyz_in (input) : the initial list of vertices, referred to in initialFaces

backroundMapping_ (input, optional) : a pointer the the mapping that represents the underlying surface

3.4 initialize

void
initialize(intArray &initialFaces, realArray &xyz_in, Mapping *backgroundMapping_ = NULL)

Purpose: Initialize an AdvancingFront with a set of initial faces and a background mapping.

initialFaces (input) : the list of initial faces (vertex lists for each face)

xyz_in (input) : the initial list of vertices, referred to in initialFaces

backroundMapping_ (input, optional) : a pointer the the mapping that represents the underlying surface

3.5 destroyInfluenceList

void
destroyInfluenceList()

Purpose: Destroy any Influences that have been created for an AdvancingFront

3.6 isFrontEmpty

bool
isFrontEmpty() const

Purpose: Returns true if the front is empty, false otherwise

3.7 insertFace

int
insertFace(const IntegerArray &vertexIDs, int z1, int z2)

Purpose: Insert a 2D face into the front

vertexIDs (input) : indices for the vertices in the face

z1, z2 (input) : elements on either side of the face /

Comments :

```
X1 +  v1
   .
   .
   .---> N
   .
   .
X2 + v2
```

X1, X2, and N are vectors The vertices should be ordered such that the front will grow in the correct direction. This means, in 2D, that X2-X1 .cross. N sticks up out of the plane using the right-hand-rule.

3.8 advanceFront

int
advanceFront(int nSteps = 1)

Purpose: Advance the front (ie grow the mesh)

nSteps (input) : attempt to grow the mesh nSteps times

3.9 addInfluence

void
addInfluence(Influence &influence)

Purpose: Add a mesh growth influence to the front (influences mesh stretching and direction)

influence (input) : a reference to the influence that needs to be added

3.10 advanceFront

const intArray
generateElementList()

Purpose: generate a list of the vertices in each element

4 Geometric Searching

and this subsection

4.1 Class GeometricADT

4.2 Constructor

GeometricADT(int rangeDimension_)

Purpose : construct a geometric search ADT for a given dimension

rangeDimension_ (input) : dimension of the physical search space

4.3 Constructor

GeometricADT(const realArray & boundingBox_)

Purpose : construct a geometric search ADT given a bounding box domain

boundingBox_ (input) : bounding box of the ADT search space

4.4 initTree

void
initTree(const realArray & boundingBox_)

Purpose : initialize the tree given a particular bounding box

boundingBox_ : bounding box for the geometric search tree

4.5 insert

int
insert(iterator &insParent, int leaf, GeomADTTuple &data)

Purpose : insert a geometric entity into the tree

insParent (input) : the position in the tree to insert the data

leaf (input) : leaf of insertParent to add the data

data (input) : data to add to the tree

4.6 addElement

int
addElement(realArray &coords, int id)

Purpose : insert id into the search tree if id give the bounding box coordinates coords

coords (input) : bounding box coordinates for id (x1min, x1max, x2min, x2max,..., xnmin, xnmax)

id (input) : id to store at the given location (probably should be a templatized type

4.7 delElement

int
delElement(iterator &delItem)

Purpose : delete an element, at location delItem, from the tree

delItem : iterator pointing to the location to delete from the tree

4.8 verifyTree

void
verifyTree()

Purpose : verify the structure and logic of the tree, usually done after deletions This method will throw a VerificationError exception if the data structure has been corrupted. Use of this method aught to be surrounded by a try block.

4.9 Class GeometricADT::iterator

Creation of a `GeometricADT::iterator` requires the specification of a target representing a location in bounding box space. The iterator will then iterate from the root of a GeometricADT to the terminal leaf that would store an object with the given target. The default constructor is protected since the iterator requires the context of both a GeometricADT and a target bounding box.

4.10 Constructor

iterator(const GeometricADT &gADT, realArray & target_)

gADT (input) : the `GeometricADT` through which to iterate

target_ (input) : a `realArray` containing the target bounding box location

4.11 Copy Constructor

iterator(iterator &x)

x : iterator to copy

4.12 isTerminal

bool
isTerminal()

4.13 Assignment to Another Iterator

iterator &
operator=(iterator &i)

i : iterator to assign to

4.14 Assignment to a Node in the Seach Tree

iterator &
operator=(ADTType &x)

x : node to assign current value to

4.15 Prefix Increment

iterator &
operator++()

4.16 Postfix Increment

iterator
operator++(int)

4.17 Class GeometricADT::traversor

A `GeometricADT::traversor` will traverse it's `GeometricADT` yielding all the elements that overlap the target bounding box. This class is the basic tool used for performing geometric searches using the `GeometricADT` class.

5 Class NTreeNode<int degree, Class Data>

5.1 Default Constructor

template<int degree, class Data>
inline
NTreeNode<degree,
NTreeNode()

Purpose : build and initialize a tree node

5.2 Constructor

template<int degree, class Data>
inline
NTreeNode<degree,
NTreeNode(Data &data_)

Purpose : build and initialize a tree node given the data for the node to contain

data (input) : reference to the data to be stored in the node

5.3 Constructor

template<int degree, class Data>
inline
NTreeNode<degree,
NTreeNode(Data &data_, NTreeNode *trunk_)

Purpose : build and initialize a tree node

data_ (input) : reference to the data to be stored in the node

trunk_ (input) : pointer to the trunk

5.4 Destructor

template<int degree, class Data>
inline
NTreeNode<degree,
~NTreeNode()

Purpose : destroy a node and all its leaves

5.5 add

template<int degree, class Data>
inline
int
NTreeNode<degree,
add(Data &data)

Purpose : add a data leaf to the next available leaf position

data (input) : data item to be stored

Returns : 0 on success

Throws :
- TreeDegreeViolation : if the tree bookkeeping is corrupt
- NodeFullError : if all the leaves in this node are ful

5.6 add

template<int degree, class Data>
inline
int
NTreeNode<degree,
add(int d, Data &data)

Purpose : add a data leaf to a specified leaf

d (input) : leaf to store data

data (input) : data item to be stored

Returns : 0 on success

Throws :
- TreeDegreeViolation : if d is not a valid leaf number (if d is greater than the degree of the tree or less than zero
- NodeFullError : if d is already used up

5.7 del

template<int degree, class Data>
inline
int
NTreeNode<degree,
del(int nDel)

Purpose : delete a specified leaf

nDel (input) : leaf to delete

Returns : 0 on success

Throws :
- TreeDegreeViolation : if d is not a valid leaf number (if d is greater than the degree of the tree or less than zero

5.8 change

template<int degree, class Data>
inline
int
NTreeNode<degree,
change(NTreeNode<degree, Data> *nPtr)

Purpose : change a node's trunk

nPtr (input) : pointer to the new trunk

Returns : 0 on success

Throws : nothing

5.9 change

template<int degree, class Data>
inline
int
NTreeNode<degree,
change(int d, NTreeNode<degree, Data> *nPtr)

Purpose : changes a particular leaf

d (input) : leaf to change

nPtr (input) : pointer to the new leaf

Returns : 0 on success

Throws :
- TreeDegreeViolation : if d is not a valid leaf number

5.10 change

template<int degree, class Data>
inline
bool
NTreeNode<degree,
querry(int d)

Purpose : see if a particular leaf has data

d (input) : leaf to querry

Returns : false if leaf d is NULL, true otherwise

Throws :
- TreeDegreeViolation : if d is not a valid leaf number

5.11 change

template<int degree, class Data>
inline
bool
NTreeNode<degree,
querry()

Purpose : see if the trunk has data

d (input) : leaf to querry

Returns : false if the trunk pointer is NULL, true otherwise

Throws :
- TreeDegreeViolation : if d is not a valid leaf number

5.12 getTrunk

template<int degree, class Data>
inline
NTreeNode<degree,Data> &
NTreeNode<degree,
getTrunk()

Purpose : return a reference to the trunk node

Returns : a reference the the trunk

Throws : nothing

5.13 const getTrunk

template<int degree, class Data>
inline
const NTreeNode<degree,Data> &
NTreeNode<degree,
getTrunk() const

Purpose : return a const reference to the trunk node

Returns : a const reference the the trunk

Throws : nothing

5.14 const getLeaf

template<int degree, class Data>
inline
NTreeNode<degree,Data> &
NTreeNode<degree, getLeaf(int d)

Purpose : return a reference to a specific leaf node

d (input) : leaf to return

Returns : a reference to the requested leaf

Throws :
- TreeDegreeViolation : if d is not a valid leaf number

5.15 const getLeaf

template<int degree, class Data>
inline
const NTreeNode<degree,Data> &
NTreeNode<degree,
getLeaf(int d) const

Purpose : return a const reference to a specific leaf node

d (input) : leaf to return

Returns : a reference to the requested leaf

Throws :
- TreeDegreeViolation : if d is not a valid leaf number

6 Exception Classes

7 Composite Grid Hybrid Connectivity

7.1 Composite Grid Hybrid Connectivities

A `CompositeGridHybridConnectivity` manages the mappings between the structured and unstructured components of hybrid meshes contained in a CompositeGrid. This class enables a user to iterate through the boundary faces of an unstructured mesh and access the adjacent structured elements. The inverse is also available where the user iterates through the elements in a structured grid seeking the adjacent unstructured elements, if any exist.

7.2 Connectivity data

- gridIndex2UnstructuredVertex : maps a particular grid and index into an unstructured grid and vertex id
- unstructuredVertex2GridIndex : maps an unstructured vertex into a grid and index
- gridVertex2UnstructuredVertex : condensation of vertexIDMapping, contains all the vertices on a particular grid that are on a hybrid interface
- boundaryFaceMapping : maps the boundary element of unstructured boundary faces into a grid and zone index

7.3 Default Constructor

CompositeGridHybridConnectivity()

7.4 Initialized constructor

CompositeGridHybridConnectivity(const int &grid_,
intArray * gridIndex2UVertex_,
intArray & uVertex2GridIndex_,
intArray * gridVertex2UVertex_,
intArray & boundaryFaceMapping_)

grid (input) : the grid number in cg that contains the unstructured mesh referred to by this connectivity

gridIndex2UVertex_ (input) : an array of IntegerArrays the length of the number of grids in cg_; This data maps a structured vertex in a particular strutured grid into the corresponding unstructured vertex id, if the mapping exists (ie, if the structured vertex lies on the hybrid interface)

uVertex2GridIndex_ (input) : maps an unstructured boundary vertex into a structured grid in cg_ and the corresponding indices on that grid

gridVertex2UVertex_ (input) : an array of IntegerArrays the length of the number of grids in cg_; a condensation of vertexIDMapping, this contains a list of all the interface vertices for each grid and the unstructured vertices they map to

boundaryFaceMapping_ (input) : for each boundary face in the unstructured part of the hybrid mesh this data structure contains the grid and indices for the adjacent structured zone.

Returns : void

Throws : CompositeGridHybridConnectivityError is thrown through this method by setCompositeGridHybridConnectivity

7.5 setCompositeGridHybridConnectivity

void
setCompositeGridHybridConnectivity(const int &grid_,
intArray * gridIndex2UVertex_,
intArray & uVertex2GridIndex_,
intArray * gridVertex2UVertex_,
intArray & boundaryFaceMapping_)

with externally computed index arrays. Generally these arrays are computed during the generation of a hybrid mesh by methods in `Ugen` and `AdvancingFront`.

grid (input) : the grid number in cg that contains the unstructured mesh referred to by this connectivity

gridIndex2UVertex_ (input) : an array of IntegerArrays the length of the number of grids in cg_; This data maps a structured vertex in a particular strutured grid into the corresponding unstructured vertex id, if the mapping exists (ie, if the structured vertex lies on the hybrid interface)

uVertex2GridIndex_ (input) : maps an unstructured boundary vertex into a structured grid in cg_ and the corresponding indices on that grid

gridVertex2UVertex_ (input) : an array of IntegerArrays the length of the number of grids in cg_; a condensation of vertexIDMapping, this contains a list of all the interface vertices for each grid and the unstructured vertices they map to

boundaryFaceMapping_ (input) : for each boundary face in the unstructured part of the hybrid mesh this data structure contains the grid and indices for the adjacent structured zone.

Returns : void

Throws : – CompositeGridHybridConnectivityError

7.6 destroy

void
destroy()

Throws : nothing

7.7 getGridIndex2UVertex

const intArray &
getGridIndex2UVertex(int grid_) const

grid (input) : the requested grid

Returns : a const IntegerArray reference to the IntegerArray with grid's connectivity to the unstructured mesh

Throws : nothing (!) (but should perform a check on the validity of grid and throw an appropriate error)

7.8 getUVertex2GridIndex

const intArray &
getUVertex2GridIndex() const

Returns : a const IntegerArray reference to the unstructred mesh's connectivity to vertices in the structured grids

Throws : nothing

7.9 getGridVertex2UVertex

const intArray &
getGridVertex2UVertex(int grid_) const

grid (input) : the requested grid

Returns : a const IntegerArray reference to the IntegerArray with grid's connectivity to the unstructured mesh, but only for those verticies sitting on the structured/unstructured interface

Throws : nothing (!) (but should perform a check on the validity of grid and throw an appropriate error)

7.10 getNumberOfInterfaceVertices

int
getNumberOfInterfaceVertices(int grid_) const

grid (input) : the requested grid

Returns : int; the number of vertices on the hybrid structured/unstructured interface for `grid`

Throws : nothing (!) (but should perform a check on the validity of grid and throw an appropriate error)

7.11 getBoundaryFaceMapping

const intArray &
getBoundaryFaceMapping() const

Returns : const IntegerArray &; the boundary face mapping array, 2nd dimension is length 4 (0 - grid, 1 - i1, 2 - i2, 3 - i3)