



# Computational Aircraft Prototype Syntheses

## AIM Programming – Mesh Writing

### For ESP Rev 1.28

Bob Haimes

[bob@geocentrictech.com](mailto:bob@geocentrictech.com) or [haimes@mit.edu](mailto:haimes@mit.edu)

Geocentric Technologies LLC

## Mesh Writer AIM Plugins

- Can always write out a mesh as part of *aimCalcOutputs*
- Can use existing MeshWriters if the structures discussed below are filled out
- Can provide support for new mesh formats to the community  
See `$ESP_ROOT/src/CAPS/aim/meshWriter` for a list of supported formats

# AIM meshWriter Structures and Functions

- Structures filled by an AIM mesh generation
- Functions for initializing and filling the structures
- The library (libaimMesh.a/aimMesh.lib) must be included in the AIM so/DLL build

## Dynamically loading the mesh writer

The meshing AIM dynamically loads the appropriate so/DLL to output the mesh file in its default location. If the mesh data is resident in memory during postAnalysis, it needs be written to disk and freed. The mesh writer shared object/DLL needs to contain just the entry points: `meshExtension` & `meshWrite` (see below).

## aimMesh Structure

The complete representation of a mesh

```
typedef struct {  
    aimMeshData *meshData;  
    aimMeshRef *meshRef;  
} aimMesh;
```

**meshData** represents the mesh coordinates and connectivity

**meshRef** mapping of the boundary mesh vertexes to the interior vertexes

A mesh generation AIM is responsible for filling the complete [aimMesh](#) Structure that is passed to a `meshWriter` shared library, which is responsible for writing the data to disk. Only the `meshRef` pointer is passed via a link to an analysis AIM.

## aimMeshElemGroup Structure

Represents a group of elements of the same type

```
typedef struct {
    char          *groupName; /* name of group or NULL */
    int           ID;         /* Group ID */
    enum aimMeshElem elementTopo; /* Element topology */
    int           order;      /* order of the element (1 - Linear) */
    int           nPoint;     /* number of points defining an element */
    int           nElems;     /* number of elements in the group */
    int           *elements;  /* Element-to-vertex connectivity (1-based)
                               nElem*nPoint in length */
} aimMeshElemGroup;
```

**groupName** group identifier that may be non-unique

**ID** group identifier that may be non-unique

**elementTopo** is one of:

```
enum aimMeshElem {aimUnknownElem, aimLine, aimTri, aimQuad, aimTet,
                  aimPyramid, aimPrism, aimHex};
```

**order** polynomial degree of element

**nPoint** number of points in an element

**nElems** number of elements the group

**elements** Element-to-vertex (1-based) connectivity nElem\*nPoint in length

## aimMeshData Structure

Represents the Cartesian coordinates and element connectivity of the mesh

```
typedef double aimMeshCoords[3];
typedef int    aimMeshIndices[2];
typedef struct {
    int          dim;           /* Physical dimension: 2D or 3D */
    int          nVertex;      /* total number of vertices in the mesh */
    aimMeshCoords *verts;      /* the xyz coordinates of the vertices
                               nVertex in length */
    int          nElemGroup;    /* number of element groups */
    aimMeshElemGroup *elemGroups; /* element groups -- nElemGroup in length */
    int          nTotalElems;   /* total number of elements */
    aimMeshIndices *elemMap;    /* group,elem map in original element ordering
                               nTotalElems in length -- can be NULL */
} aimMeshData;
```

**dim** must be 2 or 3 to represent the number Physical dimensions used in verts.  
**nVertex** number of coordinates

**verts** Coordinates stored as  $verts[iv][d]$  for  $iv \in [0, nVertex)$  and  $d \in [0, dim)$ .

**nElemGroup** number of element groups

**elemGroups** group of elements with all the same type (nElemGroups in length)

**nTotalElems** total number of elements in the mesh

**elemMap** The original element ordering (*nTotalElems* in length).  $elemMap[ie][0]$  is the 0-based element group index into *elemGroups*, and  $elemMap[ie][1]$  is 0-based index of the element in the group.

## aimMeshTessMap Structure

Represents the a boundary mesh and it's mapping to the interior

```
typedef struct {  
    ego tess;          /* the EGADS Tessellation Objects (contains Body) */  
    int *map;          /* the mapping between Tessellation vertices and  
                      mesh vertices -- tess verts in length */  
} aimMeshTessMap;
```

- tess** an EGADS tessellation of the boundary
- map** mapping from global tessellation vertices to the interior mesh verticies. Use EG\_statusTessBody to get the length.

## aimMeshBnd Structure

Represents the a boundary group information

```
typedef struct {  
    char *groupName;  /* name of group or NULL */  
    int ID;           /* Group ID */  
} aimMeshBnd;
```

- groupName** a name associated with a boundary group
- ID** an identifier associated with the group

## aimMeshRef Structure

Represents the boundary of a mesh and a reference to the full mesh

```
typedef struct {  
    enum aimMeshType type;      /* type of mesh referenced */  
    int nmap;                   /* number of EGADS Tessellation Objects */  
    aimMeshTessMap *maps;       /* the EGADS Tess Object and map to mesh verts */  
    int nbnd;                   /* number of boundary groups */  
    aimMeshBnd *bnds;           /* boundary group info */  
    char *fileName;             /* full path name (no extension) for grids */  
    int _delTess;               /* internal use only, whether tess/body ego are deleted */  
} aimMeshRef;
```

**type** is one of:

```
enum aimMeshType {aimUnknownMeshType, aimAreaMesh,  
                  aimSurfaceMesh, aimVolumeMesh};
```

**nmap** number of mappings from the boundary to the interior

**maps** boundary to the interior mapping

**nbnd** number of boundary groups

**bnds** boundary group information

**fileName** absolute path to the full mesh file name without the extension

**\_delTess** Internal usage that should not be modified



## Mesh writer entry points

The following two functions are required for each dynamically loaded mesh writer. They allow the AIM mesh writer interface the ability to complete the filenames and to output the meshes. This is dynamically loadable so that new (or custom) mesh writer can be easily attached to a CAPS session.

```
const char *extension = meshExtension()
```

**extension** the file extension used for this writer

```
icode = meshWrite(void *aimInfo, aimMesh *mesh)
```

**aimInfo** the AIM context

**mesh** the mesh data structure that will be written

**icode** integer return code

## Delete previous meshes

```
icode = aim_deleteMeshes(void *aimInfo, aimMeshRef *meshRef)
```

**aimInfo** the AIM context

**meshRef** the pointer to the Mesh Reference Structure

**icode** integer return code

This should be called during the mesh writing preAnalysis to cleanup mesh files from previous invocations of the AIM instance. This is required because if the mesh file already exists, it is not (re)written in `aim_writeMeshes`.

## Query mesh existence

```
icode = aim_queryMeshes(void *aimInfo, int index, aimMeshRef *meshRef)
```

**aimInfo** the AIM context

**index** the AnalysisOut Value index to query

**meshRef** the pointer to the Mesh Reference Structure

**icode** integer return code

This call returns `CAPS_SUCCESS` if the mesh file already exists and no others are needed, if positive then this is the number of file types that need to be written via calling `aim_writeMeshes`.

## Write meshes

```
icode = aim_writeMeshes(void *aimInfo, int index,  
                        enum capsType stype, aimMesh *mesh)
```

**aimInfo** the AIM context

**index** the AnalysisOut Value index to write

**stype** ANALYSISIN or ANALYSISOUT

**mesh** the pointer to the Mesh Structure

**icode** integer return code

If meshes need to be output (see `aim_queryMeshes`), the mesh data must be populated and then written out by calling this function.

### For stype = ANALYSISIN:

This calls `writeMesh` for each name specified by the list of strings in the analysis value. The suffix “Writer” is appended to each name.

### For stype = ANALYSISOUT:

This calls `writeMesh` for each linked solver Analysis Input (as specified in the linkage).

After this call the memory allocated to fill **mesh** should be freed.

## Write a single mesh

```
icode = aim_writeMesh(void *aimInfo, const char *writerName,  
                      const char *units, aimMesh *mesh)
```

**aimInfo** the AIM context

**writerName** the string for a mesh writer (including the suffix “Writer” at the end)

**units** length units for the output mesh (may be NULL)

**mesh** the pointer to the Mesh Structure

**icode** integer return code

## Initialize aimMeshRef

```
icode = aim_initMeshRef(aimMeshRef *meshRef, enum aimMeshType type)
```

**meshRef** the aimMeshRef instance for member data initialization

**type** the AIM mesh type (aimUnknownMeshType, aimAreaMesh, aimSurfaceMesh or aimVolumeMesh)

**icode** integer return code

## Free aimMeshRef

```
icode = aim_freeMeshRef(aimMeshRef *meshRef)
```

**meshRef** the aimMeshRef instance to free member data

**icode** integer return code

## Initialize aimMeshBnd

```
icode = aim_initMeshBnd(aimMeshBnd *meshBnd)
```

**meshBnd** the aimMeshBnd instance for member data initialization

**icode** integer return code

## Initialize aimMeshData

```
icode = aim_initMeshData(aimMeshData *meshData)
```

**meshData** the aimMeshData instance for member data initialization

**icode** integer return code

## Free aimMeshData

```
icode = aim_freeMeshData(aimMeshData *meshData)
```

**meshData** the aimMeshData instance to free member data

**icode** integer return code

## Element topological dimension

```
dim = aim_elemTopoDim(enum aimMeshElem topo)
```

**topo** the aimMeshElem element type

**dim** topological dimension of the element type: 1, 2 or 3

## Add element group to aimMeshData

```
icode = aim_addMeshElemGroup(void *aimInfo, const char *groupName,  
                             int ID, enum aimMeshElem elementTopo,  
                             int order, int nPoint,  
                             aimMeshData *meshData)
```

**aimInfo** the AIM context  
**groupName** the name of the group (may be **NULL**)  
**ID** an integer group ID  
**order** the degree of the polynomial for the elements  
**nPoint** number of points in the element  
**meshData** the aimMeshData where the element group is added  
**icode** integer return code

## Add elements to aimMeshElemGroup

```
icode = aim_addMeshElem(void *aimInfo, int nElems,  
                        aimMeshElemGroup *elemGroup)
```

**aimInfo** the AIM context  
**nElems** number of elements to add to the element group  
**elemGroup** the aimMeshElemGroup where the elements are added  
**icode** integer return code

## Write meshRef to disk

```
icode = aim_storeMeshRef(void *aimInfo, const aimMeshRef *meshRef,  
                        const char *mesextension)
```

**aimInfo** the AIM context

**meshRef** the aimMeshRef instance written to disk

**mesextension** the mesh extension used by the analysis AIM

**icode** integer return code

Note: This function should be called by an analysis AIM during preAnalysis to store a meshRef instance for mesh morphing.

## Load aimMeshRef from disk

```
icode = aim_loadMeshRef(void *aimInfo, aimMeshRef *meshRef)
```

**aimInfo** the AIM context

**meshRef** the aimMeshRef instance fill from disk

**icode** integer return code

Note: This function should be called by an analysis AIM during preAnalysis to load a meshRef instance for mesh morphing.



## Morph a meshRef

```
icode = aim_morphMeshUpdate(void *aimInfo, aimMeshRef *meshRef,  
                             int numBody, ego *bodies)
```

**aimInfo** the AIM context  
**meshRef** the aimMeshRef instance to be updated  
**numBody** number of bodies  
**bodies** ego list of new bodies (*numBody* in length)  
**icode** integer return code

Note: The tessellation objects in *meshRef* are mapped to *bodies* and the boundary to interior mapping is updated.

## Create a meshRef to a local file

```
icode = aim_localMeshRef(void *aimInfo, const aimMeshRef *meshRefIn,  
                         aimMeshRef *meshRefLocal)
```

**aimInfo** the AIM context  
**meshRefIn** the input aimMeshRef instance  
**meshRefLocal** the output aimMeshRef instance with an AIM local file  
**icode** integer return code

Note: All pointers are shallow copied to meshRefLocal

## Mesh Writing

In *exercises/session13*:

- Run `session13.py` and view the resultant binary STL file:  
`myExample/Scratch/cfd/myMeshFile.bstl`  
You can use *ParaView*, but may need to rename with the file with the extension `.stl`
- Run `2bodySTL.py` and note the difference.