



# Computational Aircraft Prototype Syntheses

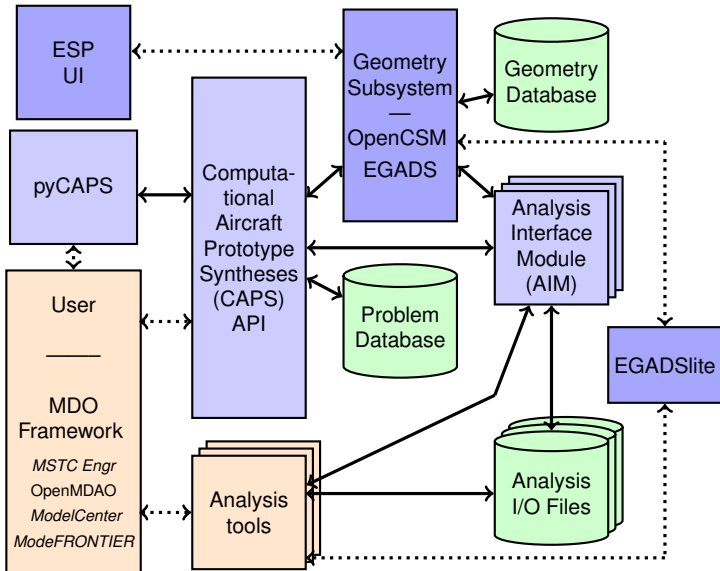
## AIM Development

### Enhanced CAPS (EnCAPS) Specification

Bob Haimes

[haimes@mit.edu](mailto:haimes@mit.edu)

Aerospace Computational Design Lab  
Massachusetts Institute of Technology



## Object-based Not *Object Orientated*

- Like *egos* in EGADS
- Pointer to a C structure – allows for a function-based API
- Treated as *blind pointers* (i.e., not meant to be dereferenced)  
Header info used to determine how to dereference the *pointer*
- API Functions
  - Returns an *int* error code or CAPS\_SUCCESS
  - Usually have one (or more) input Objects
  - Can have an output Object (usually at the end of the argument list)
- Can interface with multiple compiled languages

See \$ESP\_ROOT/doc/CAPSapi.pdf

## Problem Object

The Problem is the top-level *container* for a single mission. It maintains a single set of interrelated geometric models, analyses to be executed, connectivity and data associated with the run(s), which can be both multi-fidelity and multidisciplinary. There can be multiple Problems in a single execution of CAPS and each Problem is designed to be *thread safe* allowing for multi-threading of CAPS at the highest level.

## Value Object

A Value Object is the fundamental data container that is used within CAPS. It can represent *inputs* to the Analysis and Geometry subsystems and *outputs* from both. Also Value Objects can refer to *mission* parameters that are stored at the top-level of the CAPS database. The values contained in any *input* Value Object can be bypassed by the *linkage* connection to another Value (or *DataSet*) Object of the same *shape*. Attributes are also cast to temporary (*User*) Value Objects.

## Analysis Object

The Analysis Object refers to an instance of running an analysis code. It holds the *input* and *output* Value Objects for the instance and a directory path in which to execute the code (though no explicit execution is initiated). Multiple various analyses can be utilized and multiple instances of the same analysis can be handled under the same Problem.

## Bound Object

A Bound is a logical grouping of BRep Objects that all represent the same entity in an engineering sense (such as the “outer surface of the wing”). A Bound may include BRep entities from multiple Bodies; this enables the passing of information from one Body (for example, the aero OML) to another (the structures Body).

Dimensionally:

- 1D – Collection of Edges
- 2D – Collection of Faces

## VertexSet Object

A VertexSet is a *connected* or *unconnected* group of locations at which discrete information is defined. Each *connected* VertexSet is associated with one Bound and a single *Analysis*. A VertexSet can contain more than one DataSet. A *connected* VertexSet can refer to 2 differing sets of locations. This occurs when the solver stores it's data at different locations than the vertices that define the discrete geometry (i.e. cell centered or non-isoparametric FEM discretizations). In these cases the solution data is provided in a different manner than the geometric.

## DataSet Object

A DataSet is a set of engineering data associated with a VertexSet. The rank of a DataSet is the (user/pre)-defined number of dependent values associated with each vertex; for example, scalar data (such as *pressure*) will have rank of one and vector data (such as *displacement*) will have a rank of three. Values in the DataSet can either be deposited there by an application or can be computed (via evaluations, data transfers or sensitivity calculations).

Object	SubTypes	Parent Object
capsProblem	Parametric, Static	
capsValue	GeometryIn, GeometryOut, Branch, Parameter, User	capsProblem, capsValue
capsAnalysis		capsProblem
capsValue	AnalysisIn, AnalysisOut	capsAnalysis, capsValue
capsBound		capsProblem
capsVertexSet	Connected, Unconnected	capsBound
capsDataSet	User, Analysis, Interpolate, Conserve, Builtin, Sensitivity	capsVertexSet

Body Objects are EGADS Objects (egos)

Filtering the active CSM Bodies occurs at two different stages, once in the CAPS framework, and once in the AIMs. The filtering in the CAPS framework creates sub-groups of Bodies from the CSM stack that are passed to the specified AIM. Each AIM instance is then responsible for selecting the appropriate Bodies from the list it has received.

The filtering is performed by using two Body attributes: “capsAIM” and “capsIntent”.

## Filtering within AIM Code

Each AIM can adopt it's own filtering scheme for down-selecting how to use each Body it receives. The “capsIntent” string is accessible to the AIM, but it is for information only.



## CSM AIM targeting: “capsAIM”

The CSM script generates Bodies which are designed to be used by specific AIMs. The AIMs that the Body is designed for is communicated to the CAPS framework via the “capsAIM” string attribute. This is a semicolon-separated string with the list of AIM names. Thus, the CSM author can give a clear indication to which AIMs should use the Body. For example, a body designed for a CFD calculation could have:

```
ATTRIBUTE capsAIM $su2AIM;fun3dAIM;cart3dAIM
```

## CAPS AIM Instantiation: “capsIntent”

The “capsIntent” Body attribute is used to disambiguate which AIM instance should receive a given Body targeted for the AIM. An argument to `caps_load` accepts a semicolon-separated list of keywords when an AIM is instantiated in CAPS/pyCAPS. Bodies from the “capsAIM” selection with a matching string attribute “capsIntent” are passed to the AIM instance. The attribute “capsIntent” is a semicolon-separated list of keywords. If the string to `caps_load` is **NULL**, all Bodies with a “capsAIM” attribute that matches the AIM name are given to the AIM instance.

- Hides all of the individual Analysis details (and peculiarities)
  - Individual plugin functions *translate* from the Analysis' perspective back and forth to CAPS
  - Provides a direct connection to BRep geometry and attribution through EGADS
- Outside the CAPS Object infrastructure
  - Use of C structures
  - AIM Utility library (with the *context* embedded in `aimInfo`)

## Update Notes:

- ① Changing directories within an AIM function is no longer needed. When an AIM function is invoked, you will be in the correct location. Therefore the path has been removed from the argument list of many AIM functions. This includes `aimPreAnalysis`, `aimExecute`, `aimPostAnalysis`, `aimCalcOutput` and `aimTransfer`.
- ② There is no longer an AIM parent/child relationship. This is now accomplished via linking AnalysisOut Values of the *parent* to AnalysisIn Values of the *child*.
- ③ During restart only “Post” is executed at the last use of the AIM.
- ④ AIM specific storage is no longer indexed by the instance and is held internally.

- An AIM plugin is required for each Analysis code at:
  - a specific *intent*
  - a specific *mode* (i.e., where the inputs may be different)
- AIMs can be hierarchical
  - Parent Analysis Objects specified at CAPS Analysis load
  - Parent and child AIMs can directly communicate
- Dynamically loaded at runtime – extendibility and extensibility
  - Windows** Dynamically Loaded Libraries (name.dll)
  - LINUX** Shared Objects (name.so)
  - MAC** Bundles, CAPS will use the so file extension
- Plugin names must be unique – loaded by the name
- † indicates memory handled by CAPS in the following functions  
i.e., CAPS will free these memory blocks when necessary

The **capsValue** Structure is simply the data found within a CAPS Value Object. `aimInputs` and `aimOutputs` must fill the structure with the *type*, *form* and optionally *units* of the data. `aimInputs` also sets the default value(s) in the *vals* member. The structure's members listed below must be filled (most have defaults).

## Value Type – no default

The value *type* can be one of:

```
enum capsValueType {Boolean, Integer, Double, String, Tuple, Pointer, DoubleDot, Value};
```

**Note:** The **Pointer** type is only supported at the AIM level to communicate between AIMS. Linkages should be used from *AnalysisOut* to *AnalysisIn* to make the connection. The `units` member of the Value Structure must match for a successful link.

## The tuple structure

```
typedef struct {  
    char *name;           /* the name */  
    char *value;          /* the value for the pair */  
} capsTuple;
```

## Shape of the Value – 0 is the default

*dim* can be one of:

- 0 scalar only
- 1 vector or scalar
- 2 scalar, vector or 2D array

## Value Dimensions – 1 is the default

*nrow* and *ncol* set the dimension of the Value. If both are 1 this has a `scalar` shape. If either *nrow* or *ncol* are one then the shape is `vector`. If both are greater than 1 then this represents a 2D array of values.

## Other enumerated constants

```
enum capsFixed      {Change, Fixed};  
enum capsNull       {NotAllowed, NotNull, IsNull, IsPartial};  
enum capstMethod    {Copy, Integrate, Average};
```

### Varying Length – the default is “Fixed”

The member *lfixed* indicates whether the length of the Value is allowed to change.

### Varying Shape – the default is “Fixed”

The member *sfixed* indicates whether the *shape* of the Value is allowed to change.

### Can Value be NULL? – the default is “NotAllowed”

The member *nullVal* indicates whether the Value is or can be **NULL**  
Options are found in enum capsNULL

## capsValue Member Usage Notes

- *sfixed & dim*

If the shape is “Fixed” then *nrow* and *ncol* must fit that shape (or a lesser dimension). [Note that the length can change if *lfixed* is “Change”.] If *sfixed* is “Change” then you change *dim* before changing *nrow* and *ncol* to a higher dimension than the current setting.

- *lfixed & nrow/ncol*

If the length is “Fixed” then all updates of the Value(s) must match in both *nrow* and *ncol* (which presumes a “Fixed” shape).

- *nullVal & nrow/ncol*

*nrow* and *ncol* should remain at their values even if the Value is **NULL** to maintain the dimension (and possibly length) when “Fixed”. To indicate a **NULL** all that is necessary is to set *nullVal* to “IsNull”. The actual allocated storage can remain in the *vals* member or set to **NULL**.

- Use `EG_alloc` to allocate any memory required for the *vals* member.

```
/*
 * structure for derivative data w/ CAPS Value structure
 *   only used with "real" (double) data and
 *   only with GeometryOut or AnalysisOut Value Objects
 */

typedef struct {
    char    *name;                /* the derivative with respect to */
                                   /* including optional [n] or [n,m] for vectors/arrays */
    int     rank;                 /* the number of members in the derivative */
    double *dot;                  /* the dot values -- rank*length in length */
} capsDot;

/*
 * structure for CAPS object -- VALUE
 */
```



```

typedef struct {
    int         type;           /* value type -- capsvType */
    int         length;        /* number of values */
    int         dim;           /* the dimension */
    int         nrow;          /* number of rows */
    int         ncol;          /* the number of columns */
    int         lfixed;        /* length is fixed -- capsFixed */
    int         sfixed;        /* shape is fixed -- capsFixed */
    int         nullVal;       /* NULL handling -- capsNull */
    int         pIndex;        /* parent index */
    int         gInType;       /* 0 -- normal, 1 -- OCSM_CFGPMTR */
    union {
        int         integer;    /* single int -- length == 1 */
        int         *integers;  /* multiple ints */
        double       real;      /* single double -- length == 1 */
        double       *reals;    /* multiple doubles */
        char         *string;    /* character string (no single char) */
        capsTuple    *tuple;     /* tuple (no single tuple) */
        capsObject    *object;   /* single object */
        capsObject    **objects; /* multiple objects */
        void         *AIMptr;    /* single pointer only */
    } vals;
    union {
        int         ilims[2];   /* integer limits */
        double       dlims[2];  /* double limits */
    } limits;
    char         *units;        /* the units -- "PATH" for strings converts slashes */
    capsObject    *link;        /* the linked object (or NULL) */
    int         linkMethod;     /* the link method -- capstMethod */
    int         *partial;       /* NULL or vector/array element NULL handling */
    int         ndot;           /* the number of derivatives */
    capsDot       *dots;        /* the derivatives associated with the Value */
} capsValue;

```

# AIM Plugin Functions

- Registration & Declaring Inputs / Outputs
- Pre-Analysis, **Analysis Execution** & Retrieving Output  
Write and read files – or – use Analyses' APIs if available
- Discrete Support – Interpolation & Integration
- Data Transfers

## Initialization Information for the AIM

```
icode = aimInitialize(int qFlag, const char *uSys, void **instStore,  
                     int *major, int *minor, int *nIn,  
                     int *nOut, int *nFields, char ***fnames,  
                     int **ranks)
```

**qFlag** – 1 indicates a query and not a new analysis instance (0 or greater)

**uSys** a pointer to a character string declaring the unit system – can be **NULL**

**instStore** a returned pointer to a block of memory to be associated with this AIM instance  
may be returned as **NULL** if no AIM state data is required

**major** the returned AIM major version number

**minor** the returned AIM minor version number

**nIn** the returned number of Inputs (minimum of 1)

**nOut** the returned number of possible Outputs

**nFields** the returned number of fields to responds to for DataSet filling

**fnames** a returned pointer to a list of character strings with the field/DataSet names †

**ranks** a returned pointer to a list of ranks associated with each field †

**icode** integer return code

## Return Analysis Inputs

```
icode = aimInputs(void *instStore, void *aimInfo, int index,  
                  char **ainame, capsValue *defval)
```

**instStore** the AIM *instance* storage – NULL if called from caps\_getInput

**aimInfo** the AIM context – NULL if called from caps\_getInput

**index** the Input index [1-nIn]

**ainame** a returned pointer to the returned Analysis Input variable name

**defval** a pointer to the filled default value(s) and units – any allocated memory will be freed

**icode** integer return code

## Return Analysis Outputs

```
icode = aimOutputs(void *instStore, void *aimInfo, int index,  
                   char **aoname, capsValue *form)
```

**instStore** the AIM *instance* storage – NULL if called from caps\_getOutput

**aimInfo** the AIM context – NULL if called from caps\_getOutput

**index** the Output index [1-nOut]

**aoname** a returned pointer to the returned Analysis Output variable name

**form** a pointer to the Value Shape & Units information – to be filled  
any actual values stored are ignored/freed

**icode** integer return code

## Is the DataSet required by aimPreAnalysis – Optional

```
icode = aimUsesDataSet(void *instStore, void *aimInfo,  
                        const char *bname, const char *dname,  
                        enum capsdMethod dMethod)
```

**instStore** the AIM *instance* storage

**aimInfo** the AIM context (used by the Utility Functions)

**bname** the Bound name

**dname** the DataSet name

**dMethod** the data method used (either *Interpolate* or *Conserve*)

**icode** integer return code – use CAPS\_NOTNEEDED if not required

### Notes:

- 1 Called at caps\_makeDataSet, when the data method used is either *Interpolate* or *Conserve*, for possible dependent VertexSets with dname
- 2 If it is dependent then the Analysis Object is made *dirty* when the DataSet needs updating

## Parse Input data & Optionally Generate Input File(s)

```
icode = aimPreAnalysis(void *instStore, void *aimInfo,  
                      capsValue *inputs)
```

**instStore** the AIM *instance* storage

**aimInfo** the AIM context (used by the Utility Functions)

**inputs** the complete suite of Analysis inputs (nIn in length)

**icode** integer return code

## Execute Analysis – Optional

```
icode = aimExecute(void *instStore, void *aimInfo, int *state)
```

**instStore** the AIM *instance* storage

**aimInfo** the AIM context (used by the Utility Functions)

**state** the returned status (0 - done, 1 - running)

**icode** integer return code

## Check on running Analysis – Optional

```
icode = aimCheck(void *instStore, void *aimInfo, int *state)
```

**instStore** the AIM *instance* storage

**aimInfo** the AIM context (used by the Utility Functions)

**state** the returned status (0 - done, 1 - running)

**icode** integer return code

## Processing after the Analysis is run – No longer Optional

```
icode = aimPostAnalysis(void *instStore, void *aimInfo, int restart,  
                        capsValue *inputs)
```

**instStore** the AIM *instance* storage

**aimInfo** the AIM context (used by the Utility Functions)

**restart** restart state (0 - normal, 1 - restart invocation)

**inputs** the complete suite of Analysis inputs – for restart (nIn in length)

**icode** integer return code

## Free up any memory the AIM has stored

```
void aimCleanup(void *instStore)
```

**instStore** the block of memory associated with a particular instance

Note:

- Called a number of times, once for each instance

## Calculate/Retrieve Output Information

```
icode = aimCalcOutput(void *instStore, void *aimInfo, int index,  
                      capsValue *val)
```

**instStore** the AIM *instance* storage

**aimInfo** the AIM context (used by the Utility Functions)

**index** the Output index [1-nOut] for this single result

**val** a pointer to the capsValue data to fill – CAPS will free any allocated memory

**icode** integer return code

Note:

- Called in a *lazy* manner, only when the output is needed (and after the Analysis is run)



## Discrete Structure – Used to define a VertexSet

The CAPS *Discrete* data structure holds the spatial discretization information for a Bound. It defines reference positions for the location of the vertices that support the geometry and optionally the positions for the data locations (if these differ). This structure can contain a homogeneous or heterogeneous collection of element types and optionally specifies match positions for conservative data transfers.

## EGADS Tessellation Object

- **Now a requirement**
- Requires triangles
- Can be constructed from an external mesh generator
  - Look at `EG_initTessBody`, `EG_setTessEdge`,  
`EG_setTessFace` & `EG_statusTessBody`
  - Made understood in CAPS by invoking `aim_newTess`

```

/* defines the element discretization type by the number of reference positions
* (for geometry and optionally data) within the element.
* simple tri: nref = 3; ndata = 0; st = {0.0,0.0, 1.0,0.0, 0.0,1.0}
* simple quad: nref = 4; ndata = 0; st = {0.0,0.0, 1.0,0.0, 1.0,1.0, 0.0,1.0}
* internal triangles are used for the in/out predicates and represent linear
* triangles in [u,v] space.
* ndata is the number of data reference positions, which can be zero for simple
* nodal or isoparametric discretizations.
* match points are used for conservative transfers. Must be set when data
* and geometry positions differ, specifically for discontinuous mappings.
* For example:

```

```

*
*      2
*     /\
*    /\
*   /\
*  0-----1
*
*      neighbors
*      tri-side  vertices
*      0         1 2
*      1         2 0
*      2         0 1
*
*      4
*     /\
*    5  3
*   /\  /\
*  6
* /\  /\
0----1----2
*
*      neighbors
*      side  vertices
*      0     1 2
*      1     2 3
*      2     3 4
*      3     4 5
*      4     5 0
*      5     0 1
*
*      nref = 7
*
*      6
*     3---.---2
*    |         |
*   7.   8   .5
*    |         |
*   0---.---1
*      4
*
*      neighbors
*      quad-side  vertices
*      0         1 2
*      1         2 3
*      2         3 4
*      3         4 0
*      4         0 1
*
*      nref = 9
*
*      3-----2
*     |         |
*     |         |
*     |         |
*    0-----1
*
*      neighbors
*      side  vertices
*      0     1 2
*      1     2 3
*      2     3 4
*      3     4 0
*      4     0 1
*
*      nref = 5

```

```
*/

typedef struct {
    int    nref;           /* number of geometry reference points */
    int    ndata;          /* number of data ref points -- 0 data at ref */
    int    nmat;           /* number of match points (0 -- match at
                           geometry reference points) */

    int    ntri;           /* number of triangles to represent the elem */
    double *gst;           /* [s,t] geom reference coordinates in the
                           element -- 2*nref in length */

    double *dst;           /* [s,t] data reference coordinates in the
                           element -- 2*ndata in length */

    double *matst;         /* [s,t] positions for match points - NULL
                           when using reference points (2*nmat long) */

    int    *tris;          /* the triangles defined by geom reference indices
                           (bias 1) -- 3*ntri in length */
} capsEleType;
```

You will usually have only a small number of element types.

```
/*
 * defines the element discretization for geometric and optional data
 * positions.
 *
 */

typedef struct {
    int    tIndex;           /* the element type index (bias 1) */
    int    eIndex;          /* element owning index -- dim 1 Edge, 2 Face */
    int    *gIndices;        /* local indices (bias 1) geom ref positions,
                             tess index -- 2*nref in length */
    int    *dIndices;        /* the vertex indices (bias 1) for data ref
                             positions -- ndata in length or NULL */

    union {
        int tq[2];          /* tri or quad (bias 1) for ntri <= 2 */
        int *poly;          /* the multiple indices (bias 1) for ntri > 2 */
    } eTris;                /* triangle indices that make up the element */
} capsElement;
```

See AIAA paper 2014-0294 on the website in Publications for a complete write-up (AIAApaper2014-0294.pdf).

```
/*
 * defines a discretized collection of Elements for a body
 *
 * specifies the connectivity based on a collection of Element Types and the
 * elements referencing the types.
 */

typedef struct {
    ego            tess;          /* tessellation object associated with the
                                   discretization */
    int            nElems;        /* number of Elements */
    capsElement    *elems;        /* the Elements (nElems in length) */
    int            *gIndices;     /* memory storage for elemental gIndices */
    int            *dIndices;     /* memory storage for elemental dIndices */
    int            *poly;        /* memory storage for elemental poly */
} capsBodyDiscr;
```

```
/*
 * defines a discretized collection of Bodies
 *
 * specifies the dimensionality, vertices, Element Types, and body discretizations.
 *
 * nPoints refers to the number of indices referenced by the geometric positions
 * in the element which may be different from nVerts which is the number of
 * positions used for the data representation in the element. For simple nodal
 * or isoparametric discretizations, nVerts is zero and verts is set to NULL.
 */
typedef struct {
    int          dim;          /* dimensionality [1-3] */
    void         *instStore;   /* analysis instance storage */
    void         *aInfo;       /* AIM info */
                                /* below handled by the AIMS: */
    int          nVerts;       /* number data ref positions or unconnected */
    double       *verts;       /* data ref positions -- NULL if same as geom */
    int          *celelem;     /* 2*nVerts (body, element) containing vert or NULL */
    int          nDtris;       /* number of triangles to plot data */
    int          *dtris;       /* NULL for NULL verts -- indices into verts */
    int          nPoints;      /* number of entries in the geom positions */
    int          nTypes;       /* number of Element Types */
    capsEleType  *types;       /* the Element Types (nTypes in length) */
    int          nBodys;       /* number of Body discretizations */
    capsBodyDiscr *bodys;      /* the Body discretizations (nBodys in length) */
    int          *tessGlobal;  /* tessellation indices to this local space
                                2*nPoints in len (bodys index, global tess index) */

    void         *ptrm;        /* pointer for optional AIM use */
} capsDiscr;
```

See \$ESP\_ROOT/doc/capsDiscr.pdf for a more complete description.

## Fill-in the Discrete data for a Bound Object – Optional

```
icode = aimDiscr(char *bname, capsDiscr *discr)
```

**bname** the Bound name

Note: all of the BRep entities are examined for the attribute **capsBound**. Any that match **bname** must be included when filling this **capsDiscr**.

**discr** the Discrete structure to fill

Note: the AIM *instance*, AIM *info* pointer and the dimensionality have been filled in before this function is invoked.

**icode** integer return code

## Frees up pointer in the Discrete Structure – Optional

```
void aimFreeDiscrPtr(void *ptrm)
```

**ptrm** the optional pointer in the Discrete Structure that needs to be freed  
will not be called if the pointer is already **NULL**

## Free the Discrete data for a Bound Object – Obsolete

```
icode = aimFreeDiscr(capsDiscr *discr)
```

## Return Element in the *Mesh* – Optional

```
icode = aimLocateElement(capsDiscr *discr, double *params,  
                        double *param, int *bIndex, int *eIndex,  
                        double *bary)
```

**discr** the input Discrete Structure

**params** the input global *parametric* space (at all of the *geometry* support positions)  
rank is the dimensionality ( $t$  for 1D,  $[u, v]$  for 2D and  $[x, y, z]$  for 3D)

**param** the input requested parametric position in **params** (dimensionality in length)

**bIndex** the returned body index in **discr** where the position was found (1 bias)

**eIndex** the returned element index in **discr** where the position was found (1 bias)

**bary** the resultant Barycentric/reference position in the element **eIndex**

**icode** integer return code



## Data Associated with the Discrete Structure – Optional

```
icode = aimTransfer(capsDiscr *discr, const char *fname, int npts,  
                   int rank, double *data, char **units)
```

**discr** the input Discrete Structure

**fname** the field name to that corresponds to the fill

**npts** the number of points to be filled

**rank** the rank of the data

**data** a pointer associated with the data to be filled ( $\text{rank} \times \text{npts}$  in length)

**units** the returned pointer to the string declaring the units †  
return **NULL** to indicate unitless values

**icode** integer return code

### Fills in the DataSet Object

## Interpolation on the Bound – Optional

```
icode = aimInterpolation(capsDiscr *discr, const char *name,  
                        int bIndex, int eIndex, double *bary,  
                        int rank, double *data, double *result)  
icode = aimInterpolateBar(capsDiscr *discr, const char *name,  
                        int bIndex, int eIndex, double *bary,  
                        int rank, double *r_bar, double *d_bar)
```

**discr** the input Discrete Structure

**name** a pointer to the input DataSet name string

**bIndex** the input target body index (1 bias) in the Discrete Structure

**eIndex** the input target element index (1 bias) in the Discrete Structure

**bary** the input Barycentric/reference position in the element eIndex

**rank** the input rank of the data

**data** values at the data (or geometry) positions

**result** the filled in results (rank in length)

**r\_bar** input  $d(\text{objective})/d(\text{result})$

**d\_bar** returned  $d(\text{objective})/d(\text{data})$

**icode** integer return code

Forward and *reverse differentiated* functions

## Element Integration on the Bound – Optional

```
icode = aimIntegration(capsDiscr *discr, const char *name,  
                      int bIndex, int eIndex, int rank,  
                      double *data, double *result)  
icode = aimIntegrateBar(capsDiscr *discr, const char *name,  
                       int bIndex, int eIndex, int rank,  
                       double *r_bar, double *d_bar)
```

**discr** the input Discrete Structure

**name** a pointer to the input DataSet name string

**bIndex** the input target body index (1 bias) in **discr**

**eIndex** the input target element index (1 bias) in **discr**

**rank** the input rank of the data

**data** values at the data (or geometry) positions – **NULL** length/area/volume of element

**result** the filled in results (**rank** in length)

**r\_bar** input  $d(\text{objective})/d(\text{result})$

**d\_bar** returned  $d(\text{objective})/d(\text{data})$

**icode** integer return code

Forward and *reverse differentiated* functions

## Communication – Optional (retained but undocumented)

```
icode = aimBackdoor(void *instStore, void *aimInfo,  
                    const char *JSONin, char **JSONout)
```

**instStore** the AIM *instance* storage

**aimInfo** the AIM context

**JSONin** a pointer to a character string that represents the inputs.

**JSONout** a returned pointer to a character string that is the output of the request.

## AIM sensitivity request – Optional

```
icode = aimSensitivity(void *instStore, void *aimInfo,  
                       const char *GIname, int irow, int icol)
```

**instStore** the AIM *instance* storage

**aimInfo** the AIM context

**GIname** a character string defining the geometric parameter

**irow** the parameter row to use – 1 bias

**icol** the parameter column to use – 1 bias

**icode** integer return code

Use `aim_sensitivity` to retrieve the sensitivity data.

# AIM Helper Functions

- provides useful functions for the AIM programmer
- gives access to CAPS Object data
- note that all function names begin with `aim_`
- if any of these functions are used, then the library must be included (`libaimUtil.a/aimUtil.lib`) in the AIM so/DLL build

## Get Problem root

```
icode = aim_getRootPath(void *aimInfo, char **fullPath)
```

**aimInfo** the AIM context

**fullPath** the file path to find the root of the Problem's directory structure  
if on Windows it will contain the drive

**icode** integer return code

Note: All other uses of *path* is relative to this point.

## Get our directory name

```
icode = aim_getDirectory(void *aimInfo, char **aimName)
```

**aimInfo** the AIM context

**aimName** the directory name in the root structure of the Problem

**icode** integer return code

## Get CAPS revision

```
void aim_capsRev(int *major, int *minor)
```

**major** the returned major revision

**minor** the returned minor revision number

## Get Bodies

```
icode = aim_getBodies(void *aimInfo, const char **intent, int *nBody,  
                     ego **bodies)
```

**aimInfo** the AIM context

**intent** the returned pointer to the capsIntent string used to filter the Bodies

**nBody** the returned number of EGADS Body Objects that match the **intent**

**bodies** the returned pointer to a list of EGADS Body/Node Objects

**icode** integer return code

## Is Node Body

```
icode = aim_isNodeBody(ego body, double *xyz)
```

**body** the EGADS Body Objects to query

**xyz** the returned XYZ of the Node (if a Node Body)

**icode** integer return code

## Units conversion

```
icode = aim_convert(void *aimInfo,  const char *inUnits,  
                    double inValue, const char *outUnits,  
                    double *outValue)
```

**aimInfo** the AIM context

**inUnits** the pointer to the string declaring the source units

**inValue** the value to be converted

**outUnits** the pointer to the string declaring the desired units

**outValue** the returned converted value

**icode** integer return code



## Units multiplication

```
icode = aim_unitMultiply(void *aimInfo, const char *inUnits1,  
                        const char *inUnits2, char **outUnits)
```

**aimInfo** the AIM context

**inUnits1** the pointer to the string declaring left units

**inUnits2** the pointer to the string declaring right units

**outUnits** the returned string units = inUnits1\*inUnits2 (freeable)

**icode** integer return code

## Units division

```
icode = aim_unitDivision(void *aimInfo, const char *inUnits1,  
                        const char *inUnits2, char **outUnits)
```

**aimInfo** the AIM context

**inUnits1** the pointer to the string declaring numerator units

**inUnits2** the pointer to the string declaring denominator units

**outUnits** the returned string units = inUnits1/inUnits2 (freeable)

**icode** integer return code

## Units invert

```
icode = aim_unitInvert(void *aimInfo, const char *inUnits,  
                      char **outUnits)
```

**aimInfo** the AIM context

**inUnits** the pointer to the string declaring units

**outUnits** the returned string units = 1/inUnits (freeable)

**icode** integer return code

## Units raise to power

```
icode = aim_unitRaise(void *aimInfo, const char *inUnits,  
                     const int power, char **outUnits)
```

**aimInfo** the AIM context

**inUnits** the pointer to the string declaring units

**outUnits** the returned string units = inUnits ^ power (freeable)

**icode** integer return code

## Name to Index lookup

```
icode = aim_getIndex(void *aimInfo, const char *name,  
                     enum capssType stype)
```

**aimInfo** the AIM context

**name** the pointer to the string specifying the name to look-up  
**NULL** returns the total number of members in the subtype

**stype** GEOMETRYIN, GEOMETRYOUT, ANALYSISIN or ANALYSISOUT

**icode** index (1 bias) or negative integer return code

## Index to Name lookup

```
icode = aim_getName(void *aimInfo, int index, enum capssType stype,  
                    const char **name)
```

**aimInfo** the AIM context

**index** the index to use (1 bias)

**stype** GEOMETRYIN, GEOMETRYOUT, ANALYSISIN or ANALYSISOUT

**name** the returned pointer to the string specifying the name

**icode** integer return code

## Get GeometryIn Type

```
icode = aim_getGeomInType(void *aimInfo, int index)
```

**aimInfo** the AIM context  
**index** the index of GEOMETRYIN (1 bias)  
**icode** integer return code – CAPS\_SUCCESS is Design, EGADS\_OUTSIDE is Configuration

## Get Discretization State

```
icode = aim_getDiscrState(void *aimInfo, const char *bname)
```

**aimInfo** the AIM context  
**bname** the Bound name  
**icode** integer return code – CAPS\_SUCCESS is clean

## Get Value Structure

```
icode = aim_getValue(void *aimInfo, int index, enum capsType stype,  
                    capsValue **value)
```

**aimInfo** the AIM context  
**index** the index to use (1 bias)  
**stype** GEOMETRYIN, GEOMETRYOUT, ANALYSISIN or ANALYSISOUT  
**value** the returned pointer to the capsValue structure

## Data Transfer from Parent AIM(s) – Obsolete

```
icode = aim_getData(void *aimInfo, char *name, enum *vtype, int *rank,  
                    int *nrow, int *ncol, void **data, char **units)
```

## Establish Linkage from Analysis or Geometry

```
icode = aim_link(void *aimInfo, const char *name, enum capsType sty,  
                 capsValue *target, int irow, int icol)
```

**aimInfo** the AIM context

**name** the requested Value Object name to link

**sty** Value subtype (GEOMETRYIN, GEOMETRYOUT, ANALYSISIN or ANALSYSOUT)

**target** the pointer from aimInputs or aim\_getValue

**irow** the row to link in the target (or 0 for all)

**icol** the column to link in the target (or 0 for all)  
if both **irow** and **icol** are 0 the entire Object is linked

**icode** integer return code

## Set Tessellation for a Body – Obsolete

```
icode = aim_setTess(void *aimInfo, ego object)
```

## Register a New Tessellation

```
icode = aim_newTess(void *aimInfo, ego tess)
```

**aimInfo** the AIM context

**tess** the EGADS Tessellation Object to register

**icode** integer return code

### Notes:

- 1 If the Body associated with **tess** already has a registered Tessellation Object, the previous tessellation **ego** will be deleted
- 2 Any Tessellation Object registered will be deleted by CAPS before a Geometry regeneration
- 3 If the Body associated with **tess** is not on the OpenCSM stack, the Body Object will be deleted when the Tessellation Object is cleaned up (i.e., CAPS takes ownership of the Body from the AIM)

## Retrieve Tessellations for a Bound

```
icode = aim_getTessBound(void *aimInfo, const char *bname, ego **tess,  
                        int *nTessface, int **tessfaces)
```

**aimInfo** the AIM context

**bname** the Bound name

**tess** returned pointer to a list of tessellations associated with the Bound (freeable)

**nTessface** returned number of tessellation-face pairs

**tessfaces** returned pointer to a list of tessellation-face index pairs (freeable)  
2\*nTessface in length

**icode** integer return code

### Notes:

- 1 The tessellation-face index pairs are 1-biased where the first of the pair is the index into **tess** and the second is the Face index in the associated Body
- 2 If **nTessface** is zero then both **tess** and **tessfaces** will be **NULL**

## Get Geometry State WRT the Analysis

```
icode = aim_newGeometry(void *aimInfo)
```

**aimInfo** the AIM context

**icode** CAPS\_SUCCESS for new, CAPS\_CLEAN if not regenerated since last here

## Get the number of instances in the Analysis

```
icode = aim_numInstance(void *aimInfo)
```

**aimInfo** the AIM context

**icode** Error code (negative) or the number of instances

## Get the instance index for the Analysis

```
icode = aim_getInstance(void *aimInfo)
```

**aimInfo** the AIM context

**icode** Error code (negative) or the Instance index (Bias 0)



## Get Discretization Structure

```
icode = aim_getDiscr(void *aimInfo, const char *bname,  
                    capsDiscr **discr)
```

**aimInfo** the AIM context  
**bname** the Bound name  
**discr** pointer to the returned Discrete structure  
**icode** integer return code

## Get Data from Existing DataSet

```
icode = aim_getDataSet(capsDiscr *discr, const char *dname,  
                      enum capsdMethod *method, int *npts,  
                      int *rank, double **data)
```

**discr** the input Discrete Structure  
**dname** the requested DataSet name  
**method** the returned method used for data transfers  
**npts** the returned number of points in the DataSet  
**rank** the returned rank of the DataSet  
**data** a returned pointer to the data within the DataSet  
**icode** integer return code

## Get Bound Names

```
icode = aim_getBounds(void *aimInfo, int *nBname, char ***bnames)
```

**aimInfo** the AIM context

**nBname** returned number of Bound names

**bnames** returned pointer to list of Bound names (freeable)

**icode** integer return code

## Get Unit System

```
icode = aim_unitSys(void *aimInfo, char **unitSys)
```

**aimInfo** the AIM context

**unitSys** a returned pointer to a character string declaring the unit system – can be **NULL**

**icode** integer return code

## Clear AIM's directory

```
icode = aim_clear(void *aimInfo)
```

**aimInfo** the AIM context

**icode** integer return code

## Get Value Attributes

```
icode = aim_valueAttrs(void *aimInfo, int index, enum capssType stype,  
                      int *nValue, char ***names, capsValue **values)
```

**aimInfo** the AIM context

**index** the index to use (1 bias)

**stype** GEOMETRYIN, GEOMETRYOUT, ANALYSISIN or ANALYSISOUT

**nValue** returned number of attributes

**names** the returned names – nValue in length (freeable)

**values** the returned pointer to the capsValue structures – nValue in length (freeable)

**icode** integer return code

Note: use `EG_freeAttrs` to free up the memory.

## Get Analysis (our) Attributes

```
icode = aim_analysisAttrs(void *aimInfo, int *nValue, char ***names,  
                           capsValue **values)
```

**aimInfo** the AIM context

**nValue** returned number of attributes

**names** the returned names – nValue in length (freeable)

**values** the returned pointer to the capsValue structures – nValue in length (freeable)

**icode** integer return code

Note: use `EG_freeAttrs` to free up the memory.

## Free Attribute storage

```
void aim_freeAttrs(int nValue, char **names, capsValue *values)
```

**aimInfo** the AIM context

**nValue** the number of attributes

**names** the names to be freed – nValue in length

**values** the pointer to the capsValue structures – nValue in length

## Get Value Attributes

```
icode = aim_valueAttrs(void *aimInfo, int index, enum capsType type,  
                       int *nValue, char ***names, capsValue **values)
```

**aimInfo** the AIM context

**index** the index to use (1 bias)

**stype** GEOMETRYIN, GEOMETRYOUT, ANALYSISIN or ANALYSISOUT

**nValue** returned number of attributes

**names** the returned names – nValue in length (freeable)

**values** the returned pointer to the capsValue structures – nValue in length (freeable)

## Get Analysis (our) Attributes

```
icode = aim_analysisAttrs(void *aimInfo, int *nValue, char ***names,  
                          capsValue **values)
```

**aimInfo** the AIM context

**nValue** returned number of attributes

**names** the returned names – nValue in length (freeable)

**values** the returned pointer to the capsValue structures – nValue in length (freeable)

## Setup for Sensitivities

```
icode = aim_setSensitivity(void *aimInfo, const char *GIname,  
                           int *irow, int *icol)
```

**aimInfo** the AIM context

**GIname** the pointer to the string that matches the *Geometry Input* Parameter name

**irow** the parameter row to use – 1 bias

**icol** the parameter column to use – 1 bias

**icode** integer return code

### Notes:

- 1 **aim\_newTess** must have been invoked sometime before calling this function to set the tessellations for the Bodies of interest
- 2 Call **aim\_setSensitivity** before call(s) to **aim\_getSensitivity**.

## Get Sensitivities based on Tessellation Components

```
icode = aim_getSensitivity(void *aimInfo, ego tess, int ttype,  
                           int index, int *npts, double **dxyz)
```

**aimInfo** the AIM context

**tess** the EGADS Tessellation Object

**ttype** topological type – 0 - NODE, 1 - EDGE, 2 - FACE  
*Configuration Sensitivities* – -1 - EDGE, -2 - FACE

**index** the index in the Body (associated with the tessellation) based on the *type*

**npts** the returned number of sensitivities (number of tessellation points)

**dxyz** a pointer to the returned sensitivities – 3\*npts in length (*freeable*)

**icode** integer return code

Note:

- Call `aim_setSensitivity` before call(s) to `aim_getSensitivity`

## Get Global Tessellation Sensitivities

```
icode = aim_sensitivity(void *aimInfo, const char *name, int irow,  
                        int icol, ego tess, int ttype, int *npts,  
                        double **dxyz)
```

**aimInfo** the AIM context

**name** the pointer to the string that matches the *Geometry Input* Parameter name

**irow** the parameter row to use – 1 bias

**icol** the parameter column to use – 1 bias

**tess** the EGADS Tessellation Object

**ttype** topological type – -1 *Configuration Sensitivities*, 1 *Tessellation Sensitivities*

**npts** the returned number of sensitivities (number of global vertices)

**dxyz** a pointer to the returned sensitivities – 3\*npts in length (*freeable*)

**icode** integer return code

### Notes:

- 1 Used to get the tessellation sensitivities for the entire Tessellation Object
- 2 The number of points is the global number of vertices in the tessellation



## Function Status MACRO

```
AIM_STATUS(void *aimInfo, int status, ...)
```

**aimInfo** the AIM context

**status** return status from a function

**...** printf type format string and data

### Notes:

- 1 Tracks file, line, and function name backtrace information – if **status** != CAPS\_SUCCESS
- 2 Includes “goto cleanup” if **status** != CAPS\_SUCCESS

## Pseudo Code Examples

```
status = myfunc1(aimInfo, arg1, arg2);  
AIM_STATUS(aimInfo, status, cleanup)
```

```
status = myfunc2(aimInfo, arg1, arg2);  
AIM_STATUS(aimInfo, status, cleanup, "myfunc2 args %d, %d", arg1, arg2)
```

## ANALYSISIN Error Message MACRO

```
AIM_ANALYSISIN_ERROR(void *aimInfo, enum index, const char *format,  
                    ...)
```

**aimInfo** the AIM context

**index** index of ANALYSISIN

**format** printf format string

**...** printf data

Note: Tracks file, line, and function name backtrace information

### Pseudo Code Examples

```
mach = inputs[Mach-1].vals.real;  
if (mach < 0) {  
    AIM_ANALYSISIN_ERROR(aimInfo, Mach,  
                        "Mach = %f must be >= 0\n", mach);  
    status = CAPS_BADVALUE;  
    goto cleanup;  
}
```

## Error Message MACRO

```
AIM_ERROR(void *aimInfo, const char *format, ...)
```

**aimInfo** the AIM context  
**format** printf format string  
**...** printf data

Note: Tracks file, line, and function name backtrace information

## Message Add Line MACRO

```
AIM_ADDLINE(void *aimInfo, const char *format, ...)
```

**aimInfo** the AIM context  
**format** printf format string  
**...** printf data

## Pseudo Code Examples

```
status = aim_getBodies(aimInfo, &nBody, &bodies);
AIM_STATUS(aimInfo, status)

If (nBody != 1) {
    AIM_ERROR(aimInfo, "Only one body expected, but nBody = %d", nBody);
    AIM_ADDLINE(aimInfo, "This aim can only work with one body");
    status = CAPS_BADVALUE;
    goto cleanup;
}
```

## Warning Message MACRO

```
AIM_WARNING(void *aimInfo, const char *format, ...)
```

**aimInfo** the AIM context

**format** printf type format string

**...** printf data

### Notes:

- 1 Tracks file, line, and function name backtrace information
- 2 Use AIM\_ADDLINE to add additional lines

## Pseudo Code Examples

```
status = aim_getBodies(aimInfo, &nBody, &bodies);  
AIM_STATUS(aimInfo, status)
```

```
If (nBody > 1) {  
    AIM_WARNING(aimInfo, "Only one body will be used, but nBody = %d", nBody);  
    AIM_ADDLINE(aimInfo, "This aim only uses one body");  
}
```

## Informational Message MACRO

```
AIM_INFO(void *aimInfo, const char *format, ...)
```

**aimInfo** the AIM context

**format** printf type format string

**...** printf data

Notes:

- 1 Tracks file, line, and function name backtrace information
- 2 Use AIM\_ADDLINE to add additional lines

## Remove Error Message

```
aim_removeError(void *aimInfo)
```

**aimInfo** the AIM context

## Pseudo Code Example

```
status = myfunc3(aimInfo, arg1, arg2);  
if (status == CAPS_BADVALUE) {  
    aim_removeError(aimInfo);  
    /* Resolve CAPS_BADVALUE error */  
} else {  
    AIM_STATUS(aimInfo, status);  
}
```

## Memory Allocation MACROs

AIM\_ALLOC(**void** \*ptr, **size\_t** size, **type**, **void** \*aimInfo, **int** status)

AIM\_REALL(**void** \*ptr, **size\_t** size, **type**, **void** \*aimInfo, **int** status)

**ptr** pointer assigned allocation (must be **NULL** for AIM\_ALLOC)

**size** number of **type** allocations

**type** data type for the allocation

**aimInfo** the AIM context

**status** function return status

Notes:

- 1 Tracks file, line, and function name backtrace information
- 2 Includes “goto cleanup” on error and sets **status** = EGADS\_MALLOC

## Free Memory

AIM\_FREE(**void** \*ptr)

**ptr** frees pointer memory and sets **ptr** = **NULL**

## String Duplication MACRO

```
AIM_STRDUP(char *ptr, const char *str, void *aimInfo, int status)
```

**ptr** pointer assigned allocation (must be **NULL**)

**str** string for duplication

**aimInfo** the AIM context

**status** function return status

Notes:

- 1 Tracks file, line, and function name backtrace information
- 2 Includes “goto cleanup” on error and sets **status** = EGADS\_MALLOC

## Enum Name Creation MACRO

```
char *AIM_NAME (enum Name)
```

**Name** enumeration

Notes: Converts enumeration Index “Name” to a string and returns a duplicate string

## NULL Check MACRO

```
AIM_NOTNULL(char *ptr, void *aimInfo, int status)
```

**ptr** pointer checked

**aimInfo** the AIM context

**status** function return status

Notes: If **ptr == NULL**, sets **status = CAPS\_NULLVALUE** and then “goto cleanup”

## Pseudo Code Example

```
enum aimInputs {  
    Mach = 1,           /* index is 1-based */  
    NUMINPUT = Mach     /* Total number of inputs */  
};  
...  
if (index == Mach) {  
    *ainame = AIM_NAME(Mach);  
    ...  
}  
AIM_NOTNULL(*ainame, aimInfo, status);
```



## Initialize capsBodyDiscr Pointer

```
void aim_initBodyDiscr(capsBodyDiscr *discBody)
    discBody pointer to initialize
```

## Linear Triangle/Quad Element Type with Nodal Data

```
icode = aim_nodalTriangleType(capsEleType *eletype)
icode = aim_nodalQuadType(capsEleType *eletype)
    eletype element type pointer to fill
    icode integer return code
```

## Linear Triangle/Quad Element Type with Cell Data

```
icode = aim_cellTriangleType(capsEleType *eletype)
icode = aim_cellQuadType(capsEleType *eletype)
    eletype element type pointer to fill
    icode integer return code
```

## Return Element in a Linear *Mesh*

```
icode = aim_locateElement(capsDiscr *discr, double *params,  
                        double *param, int *eIndex, int *bIndex,  
                        double *bary)
```

- discr** the input Discrete Structure
- params** the input global *parametric* space (at all of the *geometry* support positions)  
rank is the dimensionality (*t* for 1D, [*u*, *v*] for 2D and [*x*, *y*, *z*] for 3D)
- param** the input requested parametric position in **params** (dimensionality in length)
- bIndex** the returned body index in **discr** where the position was found (1 bias)
- eIndex** the returned element index in **discr** where the position was found (1 bias)
- bary** the resultant Barycentric/reference position in the element **eIndex**
- icode** integer return code

## Interpolation on the Bound in a Linear *Mesh*

```
icode = aim_interpolation(capsDiscr *discr, const char *name,  
                          int bIndex, int eIndex, double *bary,  
                          int rank, double *data, double *result)  
icode = aim_interpolateBar(capsDiscr *discr, const char *name,  
                           int bIndex, int eIndex, double *bary,  
                           int rank, double *r_bar, double *d_bar)
```

**discr** the input Discrete Structure for a Linear *Mesh*

**name** a pointer to the input DataSet name string

**bIndex** the input target body index (1 bias) in the Discrete Structure

**eIndex** the input target element index (1 bias) in the Discrete Structure

**bary** the input Barycentric/reference position in the element **eIndex**

**rank** the input rank of the data

**data** values at the data (or geometry) positions

**result** the filled in results (**rank** in length)

**r\_bar** input  $d(\text{objective})/d(\text{result})$

**d\_bar** returned  $d(\text{objective})/d(\text{data})$

**icode** integer return code

Forward and *reverse differentiated* functions

## Element Integration on the Bound in a Linear *Mesh*

```
icode = aim_integration(capsDiscr *discr, const char *name,  
                       int bIndex, int eIndex, int rank,  
                       double *data, double *result)  
icode = aim_integrateBar(capsDiscr *discr, const char *name,  
                        int bIndex, int eIndex, int rank,  
                        double *r_bar, double *d_bar)
```

- discr** the input Discrete Structure for a Linear *Mesh*
- name** a pointer to the input DataSet name string
- bIndex** the input target body index (1 bias) in **discr**
- eIndex** the input target element index (1 bias) in **discr**
- rank** the input rank of the data
- data** values at the data (or geometry) positions – **NULL** length/area/volume of element
- result** the filled in results (rank in length)
- r\_bar** input  $d(\text{objective})/d(\text{result})$
- d\_bar** returned  $d(\text{objective})/d(\text{data})$
- icode** integer return code

Forward and *reverse differentiated* functions