Computational Aircraft Prototype Syntheses
CAPS Training Session 5 – AIM Development
Part of ESP Revision 1.14

Bob Haimes
haimes@mit.edu
Aerospace Computational Design Lab
Massachusetts Institute of Technology
CAPS Infrastructure in ESP

- ESP UI
- pyCAPS
- User
  - MDO Framework
    - Sorcer
    - OpenMDAO
    - ModelCenter
- Computational Aircraft Prototype Syntheses (CAPS) API
- Geometry Subsystem
  - OpenCSM
  - EGADS
- Analysis Subsystem
- Analysis Interface & Meshing (AIM)
- Problem Database
- Analysis I/O Files
- Geometry Database
- Analysis tools

MDO Framework

- Analysis tools

PyCAPS

- ESP UI

Analysis Interface & Meshing (AIM)
Object-based Not *Object Orientated*

- Like *egos* in *EGADS*
- Pointer to a C structure – allows for an 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 its 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).
## CAPS Objects

<table>
<thead>
<tr>
<th>Object</th>
<th>SubTypes</th>
<th>Parent Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>capsProblem</td>
<td>Parametric, Static</td>
<td></td>
</tr>
<tr>
<td>capsValue</td>
<td>GeometryIn, GeometryOut, Branch, Parameter, User</td>
<td>capsProblem, capsValue</td>
</tr>
<tr>
<td>capsAnalysis</td>
<td>capsProblem</td>
<td></td>
</tr>
<tr>
<td>capsValue</td>
<td>AnalysisIn, AnalysisOut</td>
<td>capsAnalysis, capsValue</td>
</tr>
<tr>
<td>capsBound</td>
<td>capsProblem</td>
<td></td>
</tr>
<tr>
<td>capsVertexSet</td>
<td>Connected, Unconnected</td>
<td>capsBound</td>
</tr>
<tr>
<td>capsDataSet</td>
<td>User, Analysis, Interpolate, Conserve, Builtin, Sensitivity</td>
<td>capsVertexSet</td>
</tr>
</tbody>
</table>

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 its 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*)

• 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:

```c
enum capsvType {Boolean, Integer, Double, String, Tuple, Value};
```

**Note:**
The Value type in a capsValue is only supported at the CAPS level and not in AIMs.

### The tuple structure

```c
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

```c
enum capsFixed    {Change, Fixed};
enum capsNull     {NotAllowed, NotNull, IsNull};
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 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 for vType = Value */
    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 -- not used in AIMs */
        capsObject **objects; /* multiple objects -- not used in AIMs */
    } vals;
    union {
        int ilims[2];      /* integer limits */
        double dlims[2];  /* double limits */
    } limits;
    char *units;          /* the units for the values */
    capsObject *link;     /* the linked object (or NULL) */
    int linkMethod;       /* the link method -- capstMethod */
} capsValue;
AIM Plugin Functions

- Registration & Declaring Inputs / Outputs
- Pre-Analysis & Retrieving Output
  Write and read files – or – use Analyses API if available
- Discrete Support – Interpolation & Integration
icode = aimInitialize(int ngIn, capsValue *gIn, int *qeFlg, const char *unitSys, int *nIn, int *nOut, int *nFields, char ***fnames, int **ranks)

ngIn the number of Geometry Input value structures

GIn a pointer to the list of Geometry Input value structures

qeFlg on Input: 1 indicates a query and not an analysis instance;
on Output: 1 specifies that the AIM executes the analysis

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

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 (-) or AIM instance counter

*nIn & nOut should not depend on the intent

Note the removal of nIntent and intent
**AIM – Initialization**

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

- **inst** the AIM *instance* index
- **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 – CAPS will free any allocated memory
- **icode** integer return code

```c
icode = aimOutputs(int inst, void *aimInfo, int index, char **aonam, capsValue *form)
```

- **inst** the AIM *instance* index
- **aimInfo** the AIM context (used by the Utility Functions)
- **index** the Output index [1-nOut]
- **aonam** 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
### Parse Input data & Optionally Generate Input File(s)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst</td>
<td>the AIM instance index</td>
</tr>
<tr>
<td>aimInfo</td>
<td>the AIM context (used by the Utility Functions)</td>
</tr>
<tr>
<td>apath</td>
<td>the filesystem path where the input file(s) are to be written</td>
</tr>
<tr>
<td>inputs</td>
<td>the complete suite of Analysis inputs (nIn in length)</td>
</tr>
<tr>
<td>errs</td>
<td>a pointer to the returned structure where input error(s) occurred – <strong>NULL</strong> no errors</td>
</tr>
<tr>
<td>icode</td>
<td>integer return code</td>
</tr>
</tbody>
</table>

Called to prepare the input to an Analysis or prepare the input and execute the Analysis (based on `qeFlg`).
Preform any processing after the Analysis is run – Optional

```
icode = aimPostAnalysis(int inst, void *aimInfo, const char *apath,
                        capsErrs **errs)
```

- **inst** the AIM instance index
- **aimInfo** the AIM context (used by the Utility Functions)
- **apath** the filesystem path where the file(s) have been written
- **errs** a pointer to the returned structure where error(s) may have occurred – NULL no errors
- **icode** integer return code

Free up any memory the AIM has stored

```
void aimCleanup()
```
Calculate/Retrieve Output Information

icode = aimCalcOutput(int inst, void *aimInfo, const char *apath,
int index, capsValue *val, capsErrs **errors)

inst  the AIM instance index
aimInfo  the AIM context (used by the Utility Functions)
apath  the filesystem path where the Analysis output file(s) should be read
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
errors  a pointer to the returned error structure where output parsing error(s) occurred
        NULL with no errors
icode  integer return code

Called in a lazy manner and 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

- Not a requirement – but useful in dealing with sensitivities
- Requires triangles
- Can be constructed from an external mesh generator
  - Look at `EG_initTessBody`, `EG_setTessEdge`, `EG_setTessFace` & `EG_statusTessBody`
  - Make it part of CSM & CAPS by `aim_setTess`
/* 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:
* neighbors
* 2
* tri-side vertices
* / \ 0 1 2
* / \ 1 2 0
* / \ 2 0 1
* 0-----1

neighbors
3-----2
quad-side vertices
| | 0 1 2
| | 1 2 3
| | 2 3 0
| 0-----1

neighbors
4-----3
side vertices
| | 0 1 2
| 1 2 3
| 2 3 4
| 3-----1

neighbors
nref = 7

neighbors
7. 8 .5 1 2 3

neighbors
nref = 9

neighbors
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 optionally data
 * positions.
 */
typedef struct {
  int bIndex;  /* the Body index (bias 1) */
  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 in the distribution for a more complete
/* defines a discretized collection of Elements
*
* specifies the connectivity based on a collection of Element Types and the
* elements referencing the types.
*/
typedef struct {
  int dim;          /* dimensionality [1-3] */
  int instance;     /* analysis instance */
  void *aInfo;      /* AIM info */
} capsDiscr;

/* below handled by the AIMs: */
int nPoints;       /* number of entries in the point definition */
int *mapping;      /* tessellation indices to the discrete space
  2*nPoints in len (body, global tess index) */
int nVerts;        /* number of data ref positions or unconnected */
double *verts;     /* data ref (3*nVerts) -- NULL if same as geom */
int *celem;        /* element containing vert (nVerts in len) or NULL */
int nTypes;        /* number of Element Types */
capsEleType *types; /* the Element Types (nTypes in length) */
int nElems;        /* number of Elements */
capsElement *elems; /* the Elements (nElems in length) */
int nDtris;        /* number of triangles to plot data */
int *dtris;        /* NULL for NULL verts -- indices into verts */
void *ptrm;        /* pointer for optional AIM use */
} capsDiscr;

See $ESP_ROOT/doc/capsDiscr.pdf for a more complete description.
**Fillin the Discrete data for a Bound Object – Optional**

icode = aimDiscr(char *tname, capsDiscr *discr)

- **tname**: the Bound name
  - Note: all of the BRep entities are examined for the attribute capsBound. Any that match tname 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 data in a Discrete Structure – Optional**

icode = aimFreeDiscr(capsDiscr *discr)

- **discr**: the Discrete Structure to have its members freed

- **icode**: integer return code
Return Element in the *Mesh* – Optional

```c
ICODE = aimLocateElement(capsDiscr *discr, double *params,
                         double *param, 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)
- **eIndex** the returned element index in the *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

```c
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 (*rank*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

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

```c
icode = aimInterpolateBar(capsDiscr *discr, const char *name, 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

- **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(objective)/d(result)

- **d_bar**  
  returned d(objective)/d(data)

- **icode**  
  integer return code

**Forward and reverse differentiated functions**
Element Integration on the Bound – Optional

icode = aimIntegration(capsDiscr *discr, const char *name, 
int eIndex, int rank, 
double *data, double *result)

icode = aimIntegrateBar(capsDiscr *discr, const char *name, 
int eIndex, int rank, 
double *r_bar, double *d_bar)

discr the input Discrete Structure
name a pointer to the input DataSet name string
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(objective)/d(result)
d_bar returned d(objective)/d(data)
icode integer return code

Forward and reverse differentiated functions
Data Transfer to Child AIM – Optional

icode = aimData(int inst, const char *name, enum *vtype, int *rank,
                 int *nrow, int *ncol, void **data, char **units)

- **inst**: the AIM instance index
- **name**: the agreed-upon data name to transfer
- **vtype**: value data type – returned
- **rank**: the rank of the data – returned (negative – child should free data)
- **nrow**: the number of rows – returned
- **ncol**: the number of columns – returned
- **data**: a void pointer associated with the data – returned
- **units**: the pointer to the string declaring the units (will be free’d by child) – returned
AIM specific Communication – Optional

icode = aimBackdoor(int inst, void *aimInfo, const char *JSONin,
                     char **JSONout)

- **inst**: the AIM instance index
- **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 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 in the AIM so/DLL build
## Get Bodies

\[ \text{icode} = \text{aim\_getBodies} (\text{void *aimInfo, char **intent, int *nBody, ego **bodies}) \]

- **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, Tessellation Objects (set by `aim\_setTess`) follow (length – 2*nBody)
- **icode**: integer return code

## Is Node Body

\[ \text{icode} = \text{aim\_isNodeBody} (\text{ego body, double *xyz}) \]

- **body**: the EGADS Body Objects to query
- **xyz**: the returned XYZ of the Node (if a Node Body)
- **icode**: integer return code
Units conversion

```c
icode = aim_convert(void *aimInfo, char *inUnits, double inValue,
                    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
### Name to Index lookup

```c
icode = aim_getIndex(void *aimInfo, char *name, enum 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

```c
icode = aim_getName(void *aimInfo, int index, enum stype, 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 Discretization State

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

- **aimInfo**: the AIM context
- **bname**: the Bound name
- **icode**: integer return code – CAPS_SUCCESS is clean

Get Value Structure

```c
icode = aim_getValue(void *aimInfo, int index, enum 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
- **icode**: integer return code
AIM Utility Library – Conversions

**Data Transfer from Parent AIM(s)**

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

- **aimInfo**: the AIM context
- **name**: the requested agreed-upon name to fill
- **vtype**: the returned value data type
- **rank**: the returned rank of the data (negative – data should be free’d when done)
- **nrow**: the returned number of rows
- **ncol**: the returned number of columns
- **data**: a returned void pointer associated with the data
- **units**: the returned pointer to the string declaring the units (should be free’d)
  - **NULL** indicates unitless values
- **rcode**: integer return code

**Notes**: All parent AIMs are queried. If none properly respond, this function returns **CAPS_NOTFOUND**. If multiple parents respond then this function returns **CAPS_SOURCEERR**. **Parents must not be dirty.**
Establish Linkage from Parent or Geometry

```c
icode = aim_link(void *aimInfo, char *name, enum stype,
                 capsValue *default)
```

- **aimInfo**  the AIM context
- **name** the requested Value Object name to link
- **stype** Value subtype (GEOMETRYIN, GEOMETRYOUT, ANALYSISIN or ANALSYSOUT)
- **default** the pointer from aimInputs
- **icode** integer return code

Note: For ANALYSISIN or ANALYSISOUT subtypes all parent Analyses are queried. If none is found in the parent hierarchy, this function returns CAPS_NOTFOUND. The query is performed from the oldest ancestor down. The first match is used.
Get Geometry State WRT the Analysis

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

- `aimInfo` the AIM context
- `icode` CAPS_SUCCESS for new, CAPS_CLEAN if not regenerated since last here

---

Set Tessellation for a Body

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

- `aimInfo` the AIM context
- `object` the EGADS Tessellation Object to use for the associated Body –or – the Body Object to remove and delete an existing tessellation
- `icode` integer return code

Note that the Body Object is part of the Tessellation Object

An error is raised when trying to set a Tessellation Object when one exists.

If the Problem is STATIC then the AIM (or CAPS application) is responsible for deleting the Tessellation Object. Otherwise removal of the Tessellation Object is controlled internally during Body operations. If a Tessellation Object is removed (no longer associated with the Body) then CAPS deletes the Tessellation Object.
Get Discretization Structure

icode = aim_getDiscr(void *aimInfo, 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, char *dname, enum *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

```c
type = 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)
- `type` - integer return code

Get Unit System

```c
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
Setup for Sensitivities

icode = aim_setSensitivity(void *aimInfo, char *GName, int *irow, int *icol)

aimInfo  the AIM context
GName    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_setTess 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

```c
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

```c
icode = aim_sensitivity(void *aimInfo, char *GIname, int irow,
                        int icol, ego tess, int *npts, double **dxyz)
```

- `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
- `tess` the EGADS Tessellation Object
- `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

Note: Used to get the tessellation sensitivities for the entire Tessellation Object. The number of points is the global number of vertices in the tessellation.
AIM Example Code

- Performs examples of value initialization
- Full source shows the filling/use of the capsDiscr structure using linear triangles
- PreAnalysis simply outputs GeometryIn, GeometryOut & the AnalysisIn values

The following are excerpts from skeletonAIM.c which is executable and can be found in $ESP_ROOT/src/CAPS/aim/skeleton.
#include <string.h>
#include <math.h>
#include "aimUtil.h"

#define DEBUG

#define CROSS(a,b,c) a[0] = (b[1]*c[2]) - (b[2]*c[1]);
                 a[1] = (b[2]*c[0]) - (b[0]*c[2]);
                 a[2] = (b[0]*c[1]) - (b[1]*c[0])

#define DOT(a,b) (a[0]*b[0] + a[1]*b[1] + a[2]*b[2])

/* AIM "local" per instance storage
   needed data should be added here & cleaned up in aimFreeDiscr */
typedef struct {
   int ngIn;
   capsValue *gIn;
} aimStorage;

/* AIM instance counter & storage */
static int nInstance = 0;
static aimStorage *instance = NULL;
int
aimInitialize(int ngIn, /*@null@*/ capsValue *gIn, int *qeFlag,
    /*@unused@*/ const char *unitSys, int *nIn, int *nOut,
    int *nFields, char ***fnames, int **ranks)
{
    int *ints, ret, flag, status, i;
    char **strs = NULL;
    aimStorage *tmp;

#define DEBUG
    printf("\n skeletonAIM/aimInitialize ngIn = %d qeFlag = %d!\n",
        ngIn, *qeFlag);
#endif

    /* on Input: 1 indicates a query and not an analysis instance */
    flag = *qeFlag;

    /* on Output: 1 specifies that the AIM executes the analysis
     * (i.e. no external executable is called)
     * 0 relies on external execution */
    *qeFlag = 1;

    /* specify the number of analysis inputs defined in aimInputs
     * and the number of analysis outputs defined in aimOutputs */
    *nIn    = 5;
    *nOut   = 1;

    /* return if "query" only */
    if (flag == 1) return CAPS_SUCCESS;

    /* specify the field variables this analysis can generate */
    *nFields = 2;
/* specify the dimension of each field variable */
ints = (int *) EG_alloc(*nFields*sizeof(int));
if (ints == NULL) {
    status = EGADS_MALLOC;
    goto cleanup;
}
ints[0] = 1;
ints[1] = 3;
*ranks = ints;

/* specify the name of each field variable */
strs = (char **) EG_alloc(*nFields*sizeof(char *));
if (strs == NULL) {
    status = EGADS_MALLOC;
    goto cleanup;
}
strs[0] = EG_strdup("scalar");
strs[1] = EG_strdup("coordinates");
for (i = 0; i < *nFields; i++)
    if (strs[i] == NULL) {
        status = EGADS_MALLOC;
        goto cleanup;
    }
*fnames = strs;

/* create our "local" storage for anything that needs to be persistent 
remember there can be multiple instances of the AIM */
if (nInstance == 0) {
    tmp = (aimStorage *) EG_alloc(sizeof(aimStorage));
} else {
    tmp = (aimStorage *) EG_reall(instance, (nInstance+1)*sizeof(aimStorage));
}
if (tmp == NULL) {
    status = EGADS_MALLOC;
    goto cleanup;
}
instance = tmp;

/* return the index for the instance generated & store our local data */
ret = nInstance;
instance[ret].ngIn = ngIn;
instance[ret].gIn = gIn;
nInstance++; /* increment the instance count */
return ret;

cleanup:
/* release all possibly allocated memory on error */
printf("skeletonAIM/aimInitialize: failed to allocate memory\n");
*nFields = 0;

EG_free(*ranks);
*ranks = NULL;
if (*fnames != NULL)
    for (i = 0; i < *nFields; i++) EG_free(strs[i]);
EG_free(*fnames);
*fnames = NULL;

return status;
int
aimInputs(int inst, /*@unused@*/ void *aimInfo, int index, char **ainame,
capsValue *defval)
{
capsTuple *tuple;
#ifdef DEBUG
    printf(" skeletonAIM/aimInputs instance = %d index = %d!\n", inst, index);
#endif
    if ((inst < 0) || (inst >= nInstance)) return CAPS_BADINDEX;

    /* fill in the required members based on the index */
    if (index == 1) {
        *ainame = EG_strdup("InputVariable");
        defval->type = Boolean;
        defval->vals.integer = false;
    } else if (index == 2) {
        *ainame = EG_strdup("skeletonAIMin");
        if (*ainame == NULL) return EGADS_MALLOC;
        defval->type = Double;
        defval->vals.real = 5.0;
        defval->units = EG_strdup("cm");
        if (defval->units == NULL) return EGADS_MALLOC;
    } else if (index == 3) {
        *ainame = EG_strdup("Mach"); // Mach number
        defval->type = Double;
        defval->nullVal = IsNull;
        defval->units = NULL;
        defval->lfixed = Change;
        defval->dim = Scalar;
    } else if (index == 4) {


*ainame = EG_strdup("Mesh_Format");
defval->type = String;
defval->vals.string = EG_strdup("AFLR3");
defval->lfixed = Change;

} else if (index == 5) {

/* an example of filling in a Tuple */
*ainame = EG_strdup("table");
if (*ainame == NULL) return EGADS_MALLOC;
tuple = (capsTuple *) EG_alloc(3*sizeof(capsTuple));
if (tuple == NULL) return EGADS_MALLOC;
tuple[0].name = EG_strdup("Entry1");
tuple[1].name = EG_strdup("Entry2");
tuple[2].name = EG_strdup("Entry3");
tuple[0].value = EG_strdup("Value1");
tuple[1].value = EG_strdup("Value2");
tuple[2].value = EG_strdup("Value3");

defval->type = Tuple;
defval->dim = Vector;
defval->nrow = 1;
defval->ncol = 3;
defval->vals.tuple = tuple;

} else {
    printf(" skeletonAIM/aimInputs: unknown input index = %d for instance = %d!\n", index, inst);
    return CAPS_BADINDEX;
}

return CAPS_SUCCESS;
}
int aimOutputs(int inst, /*@unused@*/ void *aimInfo, /*@unused@*/ int index,
               char **aoname, capsValue *form)
{
    #ifdef DEBUG
        printf(" skeletonAIM/aimOutputs instance = %d index = %d!\n", inst, index);
    #endif
    if ((inst < 0) || (inst >= nInstance)) return CAPS_BADINDEX;

    *aoname = EG_strdup("skeletonAIMout");
    if (*aoname == NULL) return EGADS_MALLOC;
    form->type = Double;

    return CAPS_SUCCESS;
}
int
aimPreAnalysis(int inst, void *aimInfo, /*@unused@*/ const char *apath,
   capsValue *inputs, capsErrs **errs)
{
    int i, n, status, nBody, state, npts;
    const char *name, *intents;
    double size, box[6], params[3];
    ego *bodies, dum, tess;
    capsValue *val;
#ifdef DEBUG
    printf("\n skeletonAIM/aimPreAnalysis instance = %d!\n", inst);
#endif

    *errs = NULL;
    if ((inst < 0) || (inst >= nInstance)) return CAPS_BADINDEX;

    /* look at the CSM design parameters */
    printf(" GeometryIn:\n");
    for (i = 0; i < instance[inst].ngIn; i++) {
      status = aim_getName(aimInfo, i+1, GEOMETRYIN, &name);
      if (status == CAPS_SUCCESS)
        printf(" %d: %s %d (%d,%d)\n", i+1, name,
               instance[inst].gIn[i].type, instance[inst].gIn[i].nrow,
               instance[inst].gIn[i].ncol);
    }
printf("\n  GeometryOut:\n");
/* look at the CSM output parameters */
n = aim_getIndex(aimInfo, NULL, GEOMETRYOUT);
for (i = 0; i < n; i++) {
  status = aim_getName(aimInfo, i+1, GEOMETRYOUT, &name);
  if (status != CAPS_SUCCESS) continue;
  status = aim_getValue(aimInfo, i+1, GEOMETRYOUT, &val);
  if (status == CAPS_SUCCESS)
    printf(" %d: %s %d (%d,%d)\n", i+1, name,
                val->type, val->nrow, val->ncol);
}

/* write out input list of values */
if (inputs != NULL) {
  printf("\n  AnalysisIn:\n");
  for (i = 0; i < 5; i++) {
    status = aim_getName(aimInfo, i+1, ANALYSISIN, &name);
    if (status != CAPS_SUCCESS) break;
    printf(" %d: %s %d (%d,%d) %s\n", i+1, name,
            inputs[i].type, inputs[i].nrow, inputs[i].ncol, inputs[i].units);
  }
}
/* create the tessellation */

status = aim_getBodies(aimInfo, &intents, &nBody, &bodies);
if (status != CAPS_SUCCESS) {
    printf(" skeletonAIM/aimPreAnalysis aim_getBodies = %d\n", status);
    return CAPS_SUCCESS;
}

if ((bodies == NULL) || (nBody == 0)) return CAPS_SUCCESS;

for (i = 0; i < nBody; i++) {
    /* if we have a tessellation -- skip it */
    if (bodies[i+nBody] != NULL) continue;

    /* tessellate with EGADS tessellation directly in this example */
    status = EG_getBoundingBox(bodies[i], box);
    if (status != EGADS_SUCCESS) {
        printf(" skeletonAIM/aimPreAnalysis EG_getBoundingBox = %d\n", status);
        continue;
    }

    size = box[3]-box[0];
    if (size < box[4]-box[1]) size = box[4]-box[1];
    if (size < box[5]-box[2]) size = box[5]-box[2];
    params[0] = 0.025*size;
    params[1] = 0.001*size;
    params[2] = 15.0;
    status = EG_makeTessBody(bodies[i], params, &tess);
    if (status != EGADS_SUCCESS) {
        printf(" skeletonAIM/aimPreAnalysis EG_makeTessBody = %d\n", status);
        continue;
    }
}
status = EG_statusTessBody(tess, &dum, &state, &npts);
if (status != EGADS_SUCCESS) {
    printf(" skeletonAIM/aimPreAnalysis EG_statusTessBody = %d\n", status);
    continue;
}
#endif DEBUG
    printf(" skeletonAIM/aimPreAnalysis tess %d has %d vertices!\n", i, npts);
#endif

    /* store the tessellation in CAPS */
    status = aim_setTess(aimInfo, tess);
    if (status != CAPS_SUCCESS)
        printf(" skeletonAIM/aimPreAnalysis aim_setTess = %d\n", status);
    }
    printf("\n");

    return CAPS_SUCCESS;
}
int
aimCalcOutput(int inst, /*@unused@*/ void *aimInfo, /*@unused@*/ const char *ap,
               /*@unused@*/ int index, capsValue *val, capsErrs **errors)
{
    #ifdef DEBUG
        int status;
        const char *name;

        status = aim_getName(aimInfo, index, ANALYSISOUT, &name);
        printf(" skeletonAIM/aimCalcOutput instance = %d index = %d %s %d!\n", inst, index, name, status);
    #endif

    *errors = NULL;
    if ((inst < 0) || (inst >= nInstance)) return CAPS_BADINDEX;
    val->vals.real = 3.1415926;

    return CAPS_SUCCESS;
}
int
aimFreeDiscr(/*@null@*/ capsDiscr *discr)
{
    int i;

    #ifdef DEBUG
    if (discr == NULL) {
        printf(" skeletonAIM/aimFreeDiscr w/ NULL!\n");
    } else {
        printf(" skeletonAIM/aimFreeDiscr instance = %d!\n", discr->instance);
    }
    #endif

    /* special call to cleanup our internal storage */
    if (discr == NULL) {
        if (nInstance != 0) {
            /* clean up any allocated data */
            /*
             */

            EG_free(instance);
        }
        nInstance = 0;
        instance = NULL;
    } else {

/* free up this capsDiscr */
if (discr->mapping != NULL) EG_free(discr->mapping);
if (discr->verts != NULL) EG_free(discr->verts);
if (discr->celem != NULL) EG_free(discr->celem);
if (discr->types != NULL) {
    for (i = 0; i < discr->nTypes; i++) {
        if (discr->types[i].gst != NULL) EG_free(discr->types[i].gst);
        if (discr->types[i].dst != NULL) EG_free(discr->types[i].dst);
        if (discr->types[i].matst != NULL) EG_free(discr->types[i].matst);
        if (discr->types[i].tris != NULL) EG_free(discr->types[i].tris);
    }
    EG_free(discr->types);
}
if (discr->elems != NULL) EG_free(discr->elems);
if (discr->dtris != NULL) EG_free(discr->dtris);
if (discr->ptrm != NULL) EG_free(discr->ptrm);
discr->nPoints = 0;
discr->mapping = NULL;
discr->nVerts = 0;
discr->verts = NULL;
discr->celem = NULL;
discr->nTypes = 0;
discr->types = NULL;
discr->nElems = 0;
discr->elems = NULL;
discr->nDtris = 0;
discr->dtris = NULL;
discr->ptrm = NULL;
}
return CAPS_SUCCESS;
Building and executing the AIM

**LINUX/OSX Build**
- cd $ESP_ROOT/src/CAPS/aim/skeleton
- make

**Windows Build**
- Startup a Visual Studio Command Prompt
- Execute %ESP_ROOT%\ESPenv.bat
- cd %ESP_ROOT%\src\CAPS\aim\skeleton
- nmake -f NMakefile

**Execution**
- python case.py
% python case.py
Initiating capsProblem
Loading file into our capsProblem

--> enter ocsmSave(filename=capsTmp.cpc)
--> enter ocsmCheck()
--> checks passed
CAPS Info: Couldn’t get symbol aimPostAnalysis in skeletonAIM
CAPS Info: Couldn’t get symbol aimData in skeletonAIM
CAPS Info: Couldn’t get symbol aimBackdoor in skeletonAIM

skeletonAIM/aimInitialize  ngIn = 5  qeFlag = 0!
skeletonAIM/aimInputs  instance = 0  index = 1!
skeletonAIM/aimInputs  instance = 0  index = 2!
skeletonAIM/aimInputs  instance = 0  index = 3!
skeletonAIM/aimInputs  instance = 0  index = 4!
skeletonAIM/aimInputs  instance = 0  index = 5!
skeletonAIM/aimOutputs  instance = 0  index = 1!
Default skeletonAIMin = 5.0
Current skeletonAIMin = 6.0
--> enter ocsmBuild(buildTo=0)

: 
: 
:
skeletonAIM/aimPreAnalysis instance = 0!

GeometryIn:
1: num 2 (1,1)
2: x 2 (8,1)
3: ymin 2 (8,1)
4: zmin 2 (8,1)
5: zmax 2 (8,1)

GeometryOut:
1: pVol 2 (1,1)
2: cVol 2 (1,1)

AnalysisIn:
1: InputVariable 0 (1,1) (null)
2: skeletonAIMin 2 (1,1) cm
3: Mach 2 (1,1) (null)
4: Mesh_Format 3 (1,1) (null)
5: table 4 (1,3) (null)
skeletonAIM/aimPreAnalysis tess 0 has 5492 vertices!

skeletonAIM/aimCalcOutput instance = 0 index = 1 skeletonAIMout 0!
Computed skeletonAIMout = 3.1415926
Closing our problem
skeletonAIM/aimCleanup!
EGADS Info: 0 Objects, 0 Reference in Use (of 922) at Close!