



Computational Aircraft Prototype Syntheses AIM Programming Bounds & The AIM Discretization Structure For ESP Rev 1.28

Bob Haimes

bob@geocentrictech.com or haimes@mit.edu

Geocentric Technologies LLC

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). This is accomplished either through interpolation or through a scheme that provides conservative transfers.

Internally the data to support transfers is held in the CAPS structure `capsDiscr` (one per Analysis instance) – the VertexSet Object. This is manipulated by CAPS proper and the AIM functions documented here.

Creating the detailed data associated with a Bound is the focus of this session.

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, Parameter, User	capsProblem
capsAnalysis		capsProblem
capsValue	AnalysisIn, AnalysisOut, AnalysisDynO	capsAnalysis
capsBound		capsProblem
capsVertexSet	Connected, Unconnected	capsBound
capsDataSet	FieldOut, FieldIn*, User, GeomSens, TessSens, Builtin	capsVertexSet

Body Objects are EGADS Objects (egos)

* A change in a FieldIn DataSet will *dirty* the Analysis Instance

DataSet Naming Conventions

- Multiple DataSets in a Bound can have the same Name
- Allows for automatic data transfers
- One *source* (from either *FieldOut* or *User Methods*)
- Reserved Names:

DSet Name	rank	Meaning	Comments
xyz	3	<i>Geometry</i> Positions	
xyzd	3	<i>Data</i> Positions	Not for vertex-based discretizations
param	2	[u,v] data for <i>Geometry</i> Positions	
paramd	2	[u,v] for <i>Data</i> Positions	Not for vertex-based discretizations
<i>GeomIn</i>	3	Sensitivity for the Geometry Input <i>GeomIn</i>	can have [<i>irow</i> , <i>icol</i>] in name

Initialization Information for the AIM

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

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**

aimInfo the AIM context – **NULL** if **qFlag == -1**

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 †

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

fInOut a returned pointer to a list of field flags (FIELDIN - input, FIELDOUT - output) †

icode integer return code

Conservation is a statement of integration

- Consistency with solvers is difficult (in general)
 - must hold onto the data required to do the integration
 - must be able to perform the integration in the same manner as the solver integrates
- How many different solver discretizations are there?
 - finite volume, finite element, ...
 - node-based or cell-based data storage
 - continuous or discontinuous formulations
 - AMR Cartesian
 - higher order FEM (e.g., continuous or discontinuous Galerkin)

Interdisciplinary Coupling

- Traditionally custom pairwise codes are required.
- CAPS goal: let CAPS provide the ability to transfer data internally.

Universal View of Solver Spatial Discretizations

Technique Used

- Gradient-based optimization that balances integrated quantities by adjusting the equivalent of “interpolation coefficients”
- Needs solver consistent *interpolation*, *integration* and their duals

Data required – take an FEM perspective

- Element type – must support heterogeneous discretizations
- Positions within an element are defined by the Barycentric coordinates (s, t)
 - geometry positions define the *geometry* of the cell
 - data positions define where *dependent variables* are stored
 - needed for cell-based, discontinuous and/or non-isoparametric discretizations
 - by default, the data is stored at the geometry locations
- Data to associate back to the owning geometry (i.e., the Face and parametric coordinates (u, v))

Definition of Element Types

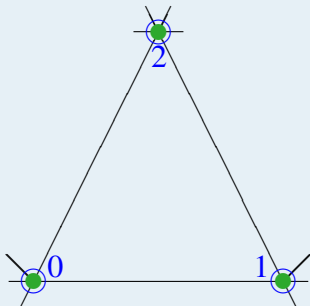
For the element examples that follow:

- **nRef** is the number of polygonal positions (per element) that the physical “corners” of the element are defined
- **nData** is the number of positions used to define the data locations in the element – 0 indicates that geometric \equiv data positions
- Higher-order positions (must be nodal, not modal) – do not contribute to the polygonal shape
 - the first positions must be those that define the polygon and should be ordered (using a right-handed traversal)
- All are indices into lists of points
 - Note: discontinuous discretizations do not share indices at bounds

Definition of Element Types

Simple continuous linear triangle

$n\text{Ref} = 3$, $n\text{Data} = 0$



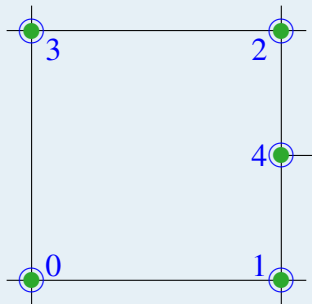
Barycentric Coordinates

geom&data	s	t
0	0	0
1	1	0
2	0	1

Definition of Element Types

Hanging vertex (AMR) quadrilateral

$nRef = 5$, $nData = 0$



Barycentric Coordinates

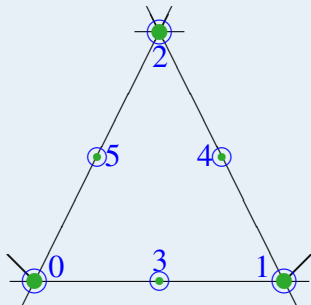
geom&data

	s	t
0	0	0
1	1	0
2	1	1
3	0	1
4	1	1/2

Definition of Element Types

Second order continuous triangle

$n\text{Ref} = 6$, $n\text{Data} = 0$



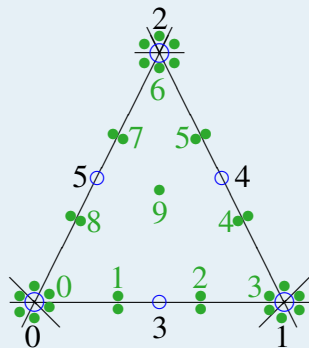
Barycentric Coordinates

$\text{geom} \& \text{data}$	s	t
0	0	0
1	1	0
2	0	1
3	1/2	0
4	1/2	1/2
5	1/2	0

Definition of Element Types

Discontinuous triangle (q=2, p=3)

nRef = 6, nData = 9



Barycentric Coordinates

geom	s	t
0	0	0
1	1	0
2	0	1
3	1/2	0
4	1/2	1/2
5	1/2	0

data	s	t	data	s	t
0	0	0	5	1/3	2/3
1	1/3	0	6	0	1
2	2/3	0	7	0	2/3
3	1	0	8	0	1/3
4	2/3	1/3	9	1/3	1/3

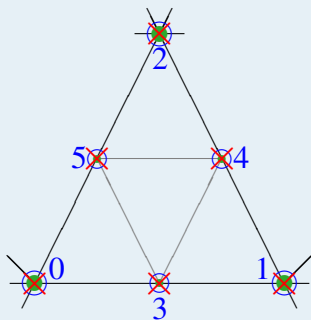
Additionally needed:

- Optional positions that support *matching* during the optimization. Required for discontinuous discretizations so that the resultant data at the reference positions are not the same.
 - **nMat** is the number of match positions
0 indicates that reference \equiv match positions
- Cutting up the element into triangles which facilitates finding a particular element given a target quilt (u, v)
 - **nTris** is the number of triangles that best represent the element in a linear sense

Extended Definition of Element Types

Second order continuous triangle

$n\text{Tris} = 4$, $n\text{Ref} = 6$
 $n\text{Data} = 0$, $n\text{Mat} = 0$



Barycentric Coordinates

geom &data	s	t
0	0	0
1	1	0
2	0	1
3	1/2	0
4	1/2	1/2
5	1/2	0

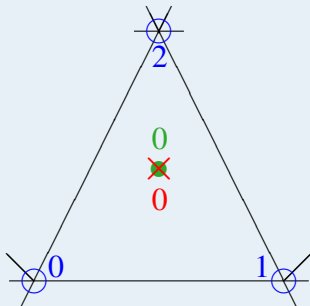
Triangle Indices

0	3	5
3	1	4
5	4	2
3	4	5

Extended Definition of Element Types

Cell-centered or discontinuous constant triangle

$n\text{Tris} = 1$, $n\text{Ref} = 3$
 $n\text{Data} = 1$, $n\text{Mat} = 1$



Barycentric Coordinates

geom	s	t
0	0	0
1	1	0
2	0	1

data

0	1/3	1/3
---	-----	-----

match

0	1/3	1/3
---	-----	-----

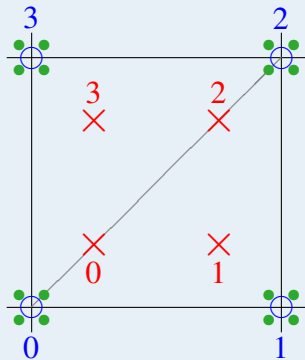
Triangle Indices

0 1 2

Extended Definition of Element Types

Discontinuous bilinear quadrilateral

$nTris = 2$, $nRef = 4$
 $nData = 4$, $nMat = 4$



Barycentric Coordinates

geom	data	s	t
0	0	0	0
1	1	1	0
2	2	1	1
3	3	0	1

match

0	1/4	1/4
1	3/4	1/4
2	3/4	3/4
3	1/4	3/4

Triangle Indices

0	1	2
0	2	3

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.

Contains enough information so that the Bound data may be visualized.

EGADS Tessellation Object

- Used to specify the discretization of the entire Body
- Requires triangles
- Can be constructed from an external mesh generator
 - Look at `EG_initTessBody`, `EG_setTessEdge`,
`EG_setTessFace` & `EG_statusTessBody`
 - Set in CAPS by invoking `aim_newTess`

Structure capsElementType

```
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 */  
    int    nseg;           /* number of element segments (sides) */  
    int    *segs;          /* the element segments by reference indices  
                           (bias 1) -- 2*nsegs in length */  
} capsElementType;
```

You will usually have only a small number of element types

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

Structure capsElement – a single element

```
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;
```

- **tIndex** – index into the collected capsEleTypes of capsDiscr
- **eIndex** – index into the owning ego in the Body
- **gIndices** – index in capsBodyDiscr member gIndices – [2*nref in length]
index in capsDiscr member tessGlobal
- **dIndices** – index in capsBodyDiscr member dIndices – [ndata in length or **NULL**]
- **eTris** – triangle index/indices from the ego of the Body tessellation
poly must be allocated in capsBodyDiscr – this is a pointer into that memory block

Structure capsBodyDiscr

```
/*
 * 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 */
    int            globalOffset;  /* tessellation global index offset across bodies */
} capsBodyDiscr;
```

- **gIndices** – allocated memory block for the collection of reference positions [$\text{length} - \sum 2 * \text{nref}$]
- **dIndices** – allocated memory block for data indices [$\sum \text{ndata}$ in length or **NULL**]
- **poly** – allocated memory block for polygon indices [$\sum \text{ntri}$ in length or **NULL**]
- **globalOffset** – an offset into `tessGlobal` that produces a unique index across multiple bodies

```

/*
 * defines a discretized collection of Bodies
 *
 * 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          *celem;        /* 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          nDsegs;        /* number of segs (sides) to plot data mesh */
    int          *dsegs;        /* 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 complete description

- The first members (`dim`, `instance` and `ainfo`) are filled by CAPS before calling `aimDiscr`.
- All physical positions (except for those in `verts`) are found in the associated Tessellation Object, which may be created in the AIM and set in CAPS by invoking `aim_setTess`.
- The number of geometric reference points (`nPoints`) is the total number of vertices that support the discretization.
- The number of elements types is set by the member `nTypes` and the types themselves are defined by a pointer to the allocated block of memory `types` which contains `nTypes` of `capsEleType`.
- The number of vertices used for the data positions is defined by `nVerts`. If `nVerts` is nonzero:
 - `nVerts` entries must be allocated for the member `verts` and this must be filled with the XYZ positions associated with the appropriate data reference positions defined as part of the elements. The member `celem` refers to the index of the element containing the position and must be allocated consistent with `verts`.
 - The number of triangles used for plotting data reference information is set by the member `nDtris`. The actual triangles are defined in `dtris`, which should be 3 times `nDtris` in length. The values stored are the indices into the `verts` member (bias 1). The number of segments (`nDsegs`) is associated with plotting the data mesh information, which is defined in `dsegs` (should be 2 times `nDsegs` in length), which contains pairs of indices into the `verts` member (bias 1).
- The association between geometric reference points and the Tessellation Object is done by the `tessGlobal` member. The first of the pair of integers in an index (bias 1) into the `bodys` member. The second is the global index (bias 1) in the Tessellation Object.
Note: the `tessGlobal` memory block is allocated and populated automatically within CAPS.
- The member `ptrm` is set aside for the plugin author and can be used to hold on to any data needed to communicate with and between the AIM routines.

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**

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

AIM Helper Functions

Discretization Structure

- 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 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, char **units)
```

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

units the unit string associated with the data within the DataSet

icode integer return code

Note: may only be called from aimPreAnalysis

Initialize capsBodyDiscr Pointer

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

Linear Triangle/Quad Element Type with Nodal Data

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

Linear Triangle/Quad Element Type with Cell Data

```
icode = aim_cellTriangleType(capsEleType *eetype)
icode = aim_cellQuadType(capsEleType *eetype)
    eetype 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

In *exercises/session12*:

- Examine the differences between `session08.py` and `session12.py` in the functions common to both
- Examine, build and execute `session12.py` as well as `crossXfer.py`
Why the differences? Why turn off `autoExec` for one of the instances?
- Review the differences between `myAIM.c` from *session08* and the source in *exercises/session12*.
- Run `sensitivity.py` which uses *Bounds* to store (and in this case view) geometric sensitivities.
Examine other **CSM** DESPMTR sensitivities.