



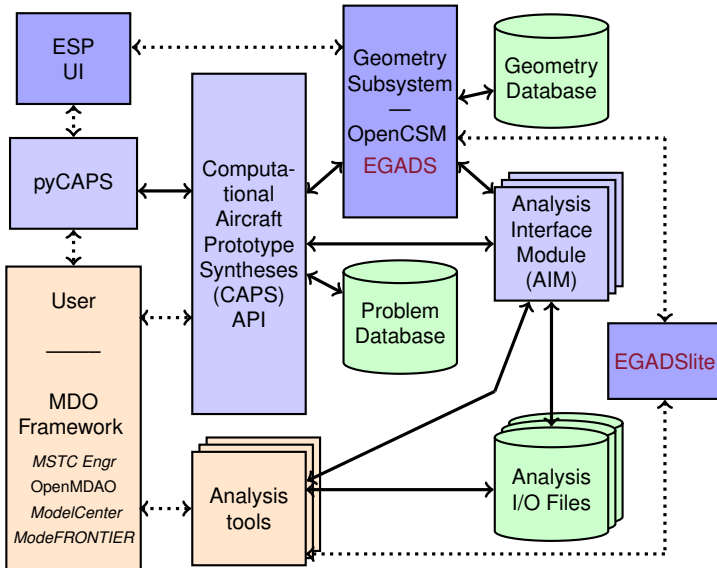
**The EGADS API**  
**Engineering Geometry Aircraft Design System**  
**at ESP Revision 1.26**

**Bob Haimes**

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

Aerospace Computational Design Lab  
Department of Aeronautics & Astronautics  
Massachusetts Institute of Technology

# Part of the Engineering Sketch Pad (ESP)



● Overview	4
● EGADS Objects	
● Geometry	16
● Topology	30
● <i>Effective Topology</i>	37
● Tessellation	40
● The Model	44
● Others	45
● EGADS & EGADSlite API	
● Utility & IO Functions	51
● Attribution	64
● Geometry	73
● Topology	88
● Tessellation	112
● Top-Down Build Functions	137
● <i>Effective Topology</i>	151
● API Index	156

## **Provide a “bottom up” and/or Constructive Solid Geometry foundation for building Aircraft (or other mechanical devices)**

- Built upon OpenCASCADE
  - Open Source solid modeling geometry kernel
  - Support for manifold and non-manifold geometry
  - Reading & writing IGES, STEP and native formats
  - C++ with ~17,000 methods!
- Open Source (LGPL v2.1)
- C/C++, FORTRAN, Python and Julia Interfaces
  - Single API with minor variations for FORTRAN
  - Always returns an integer code (success/error condition)
  - Requires C pointer access in FORTRAN
    - Cray-pointer construct –or–
    - C-pointers (2003 extension to FORTRAN 90)
    - Both supported by Intel FORTRAN and gfortran
    - API contains memory functions

## System Support

- **macOS** with clang, ifort and/or gfortran
- **LINUX** with gcc, ifort and/or gfortran
- **Windows** with Microsoft Visual Studio C++ and ifort
- No global variables and thread-safe
- Various levels of verbose output (0-none, through 3-debug)
- Written in C and C++
- Fortran bindings written in C
- pyEGADS requires no other dependencies other than a current version of Python
- jlEGADS requires Julia v1.6 or greater

## EGADS Objects

- Treated as “blind” pointers – an **ego**
  - Allows for an *Object-based* programming model
  - Can access internals in C/C++
- Context Object holds *global* information
- **egos** have:
  - *Owner*: Context, Body, EBody, or Model
  - Reference Objects (objects they depend upon)
- **egos** are INTEGER\*8 variables in FORTRAN
  - Allows for same source code regardless of size of pointer
  - Requires “freeing” of internal lists of objects (not for C/C++)

## The Context Object and Threads

- When a Context Object is created (`EG_open` – page 51) the calling thread ID is saved within the Context
- Any construction functions or functions that change the attribute storage must be done from the thread stored in the Context
- The Context's thread may be modified by invoking `EG_updateThread` (page 56) called from the new thread

## MultiThreading

- After a thread is spawned, it can call `EG_open` (page 51) to setup a Context to use with the thread
  - Will work with native threads, ESP's EMP package or *OpenMP*
- Use `EG_copyObject` (page 58) to copy an object from its owning Context to the target Context specified in the 2nd argument
- Use `EG_contextCopy` (page 59) to copy an object to the target Context specified by the 1st argument

See [\\$ESP\\_ROOT/src/EGADS/examples/multiContext.c](#) for an example using EMP

## EGADSlite – for HPC Environments

- No construction supported
- Same API and Object model as EGADS
  - Can use EGADS to prototype/build EGADSlite code
- Suitable for MPI
  - From EGADS via a *stream*, see EG\_exportModel – page 62
  - To EGADSlite from the *stream*, see EG\_importModel – page 62
  - *Stream* setup to Broadcast (or write to disk)
- ANSI C – No OpenCASCADE
- Tiny memory footprint
- Thread safe and scalable
  - EGADS' OpenCASCADE evaluation functions replaced with those written for EGADSlite

See [\\$ESP\\_ROOT/externApps/Pagoda/EGADSServer](#) for an MPI example



- Context – Holds the *globals*
- Transform
- Tessellation
- Nil (allocated but not assigned) – internal
- Empty – internal
- Reference – internal
- Geometry
  - pcurve, curve, surface
- Topology
  - Node, Edge, Loop, Face, Shell, Body, Model
- *Effective Topology*
  - EEdge, ELoop, EFace, EShell, EBody

See [\\$ESP\\_ROOT/include/egadsTypes.h](#) for a list of defines

## C structure definition - an ego

```
typedef struct egObject {  
    int      magicnumber;      /* must be properly set to validate  
                                the object */  
  
    short    oclass;           /* object class */  
    short    mtype;           /* object member type */  
    void     *attrs;           /* attributes or reference */  
    void     *blind;           /* blind pointer to OpenCASCADE or  
                                EGADS data */  
  
    struct egObject *topObj;    /* top of the hierarchy or  
                                context (if top) */  
  
    struct egObject *ref;       /* threaded list of references */  
    struct egObject *prev;      /* back pointer */  
    struct egObject *next;      /* forward pointer */  
} egObject;  
#define ego egObject*
```

## Context Object

- Start of dual threaded-list of active egos
- Pool of deleted objects

## Attribution Modes

- Single [default] – One attribute on an Object can have the **name**
- Full – There can be any number of attributes with the same **name**

## Attributes

- Are identified by a **name** (character string with no spaces or other special characters)
- Each attribute has a single **type**:
  - Integer
  - Real (double precision)
  - String (can have spaces and other special characters)
  - CSys – Coordinate System (uses the Real storage)
  - Ptr – Supplied pointer (not persistent and the programmer is responsible for memory management, i.e. freeing the storage). Uses the String pointer.
- And a **length** (for Integer, Real and CSys types)

## Objects & Attributes

- Any Object (except for Reference) may have multiple Attributes
- Only Attributes on Topological Objects are copied (except for Pointers)
- Only Attributes on Topological, *Effective* and Tessellation Objects are persistent (except for Pointer Types) – and this is available only through “.egads” file IO.

## SBOs and Intersection Functions

- Attributes on Faces will be carried through to the resultant fragments after intersections (except for Pointer types)
- Unmodified Topology maintains their attributes (except for Pointers)

## More Complex Associations

From `EG_filletBody`, `EG_chamferBody` and `EG_hollowBody` a list is returned containing an *operation* and an index to a source object in the Body:

<i>operation</i>	Description
NODEOFF (1)	The Face is the result of a Node – the index is that of the Node in the source Body
EDGEOFF (2)	The Face is the result of an Edge – the index is the Edge index (see <code>EG_indexBodyTopo</code> , page 97)
FACEDUP* (3)	The Face is an exact copy of the source
FACECUT* (4)	The Face has been trimmed or split from the source
FACEOFF (5)	The Face is offset from the source Face – the index is that of the source

\* Note: this information is redundant with the use of Face attribution

## Coordinate Systems – ATTRCSYS

- Input Reals must be one of:
  - Any Object may have 9 values
    - position<sup>†</sup>, first direction<sup>†</sup>, second direction<sup>†</sup>
  - FACE/SURFACE can also have 6 or 3 values
    - u, v, flip, second direction<sup>†</sup> (first direction is flip\*normal)
    - u, v, *idir*: 1 – udir, 2 – vdir, 3 – -udir, 4 – -vdir  
first direction is the normal, second is set by *idir*  
if *idir* is negated then the normal direction is flipped
  - EDGE/CURVE can have 5 values
    - t, flip, second direction<sup>†</sup> (first direction is flip\*tangent)
  - NODE can have 6 values
    - first direction<sup>†</sup>, second direction<sup>†</sup>
- Output is the position and 3 orthonormal directions
  - 12 doubles returned after the input values or `egads.cs.system.ortho[4][3]` for `EG_attributeGet` (page 67) and `EG_attributeRet` (page 66)

### Notes:

- 1 third direction is implied by first  $\times$  second
- 2 <sup>†</sup> transformed when object has been transformed
- 3 The actual number of doubles is the attribute length above + 12

## Full Attribution – not the default mode

- 1 Edge and Node attributes are tracked through EGADS operations.
- 2 Multiple attributes with the same **name** are maintained. This did not matter in the default scheme because rarely did Faces have this issue, and when it did happen, the attributes from the “tool” of SBOs were used.
- 3 To track multiple (same **name**) attributes attached to an object with different values, a sequence number is internally added to the **name**. In the past the **name** of the attribute could not have spaces, so a space is used as the delimiter.
- 4 When merging attributes with the same **name**:
  - a) If there is only a single attribute with the **name** and the value on the proposed merged attribute is the same then there is no merge/addition.
  - b) If the values are different, then the **name** of the existing attribute is appended with “ 1” and the **name** of the added attribute is appended the sequence number “ 2”. That is, an attribute **name** without a sequence number can only exist with a single attribute of that **name**.
  - c) If there are already multiple sequences for the same **name**, any additional merges adds the attribute and bumps the sequence –unless–
  - d) Any attribute merge that has the same value as an existing attribute (of the same **name**) will not be added. This is consistent with (4a) above for the single (non sequenced) attribute.
- 5 Sequence numbers always start with 1 and go to the number of attributes sharing that **name**. You will not find a single attribute with a sequence number (the sequence number extension will be removed by EGADS).

## oclass = PCURVE – Parameter Space Curves

- 2D curves in the Parametric space  $[u, v]$  of a surface
- Types: Line, Circle, Ellipse, Parabola, Hyperbola, Trimmed, Bezier, BSpline, Offset
- All types abstracted to  $[u, v] = g(t)$

## oclass = CURVE

- 3D curve – single running parameter  $(t)$
- Types: Line, Circle, Ellipse, Parabola, Hyperbola, Trimmed, Bezier, BSpline, Offset
- All member types abstracted to  $[x, y, z] = g(t)$



## oclass = SURFACE

- 3D surfaces of 2 parameters  $[u, v]$
- Types: Plane, Spherical, Cylindrical, Revolution, Toriodal, Trimmed, Bezier, BSpline, Offset, Conical, Extrusion
- All types abstracted to  $[x, y, z] = f(u, v)$

## Detailed Geometry

- Geometry is created by invoking `EG_makeGeometry` (page 73)
- Geometry is queried via calls to `EG_getGeometry` (page 74)
- The information is always a pointer to **doubles** with an optional pointer to **ints** – the lengths required are described below
- Some member types require an **ego** as a reference
- Analytic derivatives exist for many Geometry functions – see [\\$ESP\\_ROOT/doc/EGADS/EGADS\\_dot/EGADS\\_dot.pdf](#)

## mtype = LINE

data length $\Rightarrow$	PCurve – 4	Curve – 6
Location	$[u, v]$	$[x, y, z]$
Direction	$[dir_u, dir_v]$	$[dir_x, dir_y, dir_z]$

## mtype = CIRCLE

data length $\Rightarrow$	PCurve – 7	Curve – 10
Center	$[u, v]$	$[x, y, z]$
Xaxis	$[xax_u, xax_v]$	$[xax_x, xax_y, xax_z]$
Yaxis	$[yax_u, yax_v]$	$[yax_x, yax_y, yax_z]$
Radius		

note: Xaxis and Yaxis should be orthogonal

## mtype = ELLIPSE

data length $\Rightarrow$	PCurve – 8	Curve – 11
Location	$[u, v]$	$[x, y, z]$
Xaxis	$[xax_u, xax_v]$	$[xax_x, xax_y, xax_z]$
Yaxis	$[yax_u, yax_v]$	$[yax_x, yax_y, yax_z]$
MajorRadius		
MinorRadius		

note: Xaxis and Yaxis should be orthogonal

## mtype = PARABOLA

data length $\Rightarrow$	PCurve – 7	Curve – 10
Location	$[u, v]$	$[x, y, z]$
Xaxis	$[xax_u, xax_v]$	$[xax_x, xax_y, xax_z]$
Yaxis	$[yax_u, yax_v]$	$[yax_x, yax_y, yax_z]$
Focus		

note: Xaxis and Yaxis should be orthogonal

## mtype = HYPERBOLA

data length $\Rightarrow$	PCurve – 8	Curve – 11
Location	$[u, v]$	$[x, y, z]$
Xaxis	$[xax_u, xax_v]$	$[xax_x, xax_y, xax_z]$
Yaxis	$[yax_u, yax_v]$	$[yax_x, yax_y, yax_z]$
MajorRadius		
MinorRadius		

note: Xaxis and Yaxis should be orthogonal

## mtype = TRIMMED

data length $\Rightarrow$	PCurve – 2	Curve – 2
Range	$t\text{-start}, t\text{-end}$	$t\text{-start}, t\text{-end}$

note: Requires reference geometry of same class (PCURVE or CURVE)

## mtype = OFFSET

data length $\Rightarrow$	PCurve - 1	Curve - 4
Direction Offset	—	$[dir_x, dir_y, dir_z]$

note: Requires reference geometry of same class (PCURVE or CURVE)

## mtype = BEZIER

<b>int</b> [3]	Description
Bit Flag	2 – rational, 4 – periodic
Degree	nCP-1 up to 25 (not used on input)
nCP	number of control points

<b>doubles</b>	PCurve	Curve
Control Points	2*nCP	3*nCP
Weights*	nCP	nCP

\* note: Weights exist only if rational

mtype = BSPLINE (includes NURBS)

int [4]	Description
Bit Flag	2 – rational, 4 – periodic
Degree	
nCP	
nKnots	
	number of control points
	number of knots

doubles	PCurve	Curve
Knots	nKnots	nKnots
Control Points	2*nCP	3*nCP
Weights*	nCP	nCP

\* note: Weights exist only if rational

## mtype = PLANE

data length $\Rightarrow$	9
Location	$[x, y, z]$
Xaxis	$[xax_x, xax_y, xax_z]$
Yaxis	$[yax_x, yax_y, yax_z]$

note: Xaxis and Yaxis should be orthogonal

## mtype = SPHERICAL

data length $\Rightarrow$	10
Center	$[x, y, z]$
Xaxis	$[xax_x, xax_y, xax_z]$
Yaxis	$[yax_x, yax_y, yax_z]$
Radius	

notes:

- 1 Xaxis and Yaxis should be orthogonal
- 2 negative Radius indicates a left-handed coordinate system

mtype = CONICAL

data length $\Rightarrow$	14
Location	$[x, y, z]$
Xaxis	$[xax_x, xax_y, xax_z]$
Yaxis	$[yax_x, yax_y, yax_z]$
Zaxis	$[zax_x, zax_y, zax_z]$
Angle	
Radius	

notes:

- 1 Xaxis and Yaxis and Zaxis should all be orthogonal
- 2 Zaxis is the rotation axis and may be left-handed
- 3 Angle is in radians from 0 to  $\pi/2$



## mtype = CYLINDRICAL

data length $\Rightarrow$	13
Location	$[x, y, z]$
Xaxis	$[xax_x, xax_y, xax_z]$
Yaxis	$[yax_x, yax_y, yax_z]$
Zaxis	$[zax_x, zax_y, zax_z]$
Radius	

notes:

- 1 Xaxis and Yaxis and Zaxis should all be orthogonal
- 2 Zaxis is the rotation axis and may be left-handed

## mtype = EXTRUSION

data length $\Rightarrow$	3
Direction	$[dir_x, dir_y, dir_z]$

note: requires reference geometry of class CURVE

## mtype = TOROIDAL

data length $\Rightarrow$	14
Center	$[x, y, z]$
Xaxis	$[xax_x, xax_y, xax_z]$
Yaxis	$[yax_x, yax_y, yax_z]$
Zaxis	$[zax_x, zax_y, zax_z]$
MajorRadius	
MinorRadius	

notes:

- ① Xaxis and Yaxis and Zaxis should all be orthogonal
- ② Zaxis is the rotation axis and may be left-handed

## mtype = REVOLUTION

data length $\Rightarrow$	6
Center	$[x, y, z]$
Axis	$[axi_x, axi_y, axi_z]$

note: requires reference geometry of class CURVE

## mtype = TRIMMED

data length $\Rightarrow$	4
uRange	$[u_{min}, u_{max}]$
vRange	$[v_{min}, v_{max}]$

note: requires reference geometry of class SURFACE

## mtype = OFFSET

data length $\Rightarrow$	1
distance	

notes:

- 1 requires reference geometry of class SURFACE
- 2 Offset distance is applied normal to the reference

mtype = BEZIER

int [5]	Description
Bit Flag	2 – rational, 4 – uPeriodic , 8 – vPeriodic
uDegree	nCPu-1 up to 25 (not used on input)
nCPu	number of control points in the $u$ direction
vDegree	nCPv-1 up to 25 (not used on input)
nCPv	number of control points in the $v$ direction

doubles	packed data
Control Points	$3 * nCPu * nCPv$
Weights*	$nCPu * nCPv$

\* note: Weights exist only if rational



mtype = BSPLINE (includes NURBS)

int [7]	Description
Bit Flag	2 – rational, 4 – uPeriodic , 8 – vPeriodic
uDegree	the degree of the BSpline in the $u$ direction
nCPu	number of control points in $u$
nUKnots	number of knots in $u$
vDegree	the degree of the BSpline in the $v$ direction
nCPv	number of control points in $v$
nVKnots	number of knots in $v$

doubles	packed data
uKnots	3*nCPu*nCPv nCPu*nCPv
vKnots	
Control Points	
Weights*	

\* note: Weights exist only if rational

## Boundary Representation – BRep

<i>Top</i> <i>Down</i>    <i>Bottom</i> <i>Up</i>	Topological Entity	Geometric Entity	Function
	Model		
	Body	Solid, Sheet, Wire	
	Shell		
	Face	<b>surface</b>	$(x, y, z) = \mathbf{f}(u, v)$
	Loop	<b>pcurves*</b>	
	Edge	<b>curve</b>	$(x, y, z) = \mathbf{g}(t)$
	Node	<b>point</b>	

- Nodes that bound Edges may not be on underlying curves
- Edges in the Loops that trim a Face may not sit on the surface hence the use of pcurves
- \* Loops may be geometry free or have associated pcurves (one for each Edge) and the surface where the pcurves reside

See [\\$ESP\\_ROOT/doc/Concepts.pdf](#) for a pictorial view of EGADS Topology

## Node

- Contains a point –  $[x, y, z]$
- Member Types: **none**

## Edge

- Has a 3D curve (if not DEGENERATE)
- Has a  $t$  range ( $t_{min}$  to  $t_{max}$ , where  $t_{min} < t_{max}$ )  
Note: The positive orientation is going from  $t_{min}$  to  $t_{max}$
- Has a Node for  $t_{min}$  and for  $t_{max}$  – can be the same Node
- Member Types:
  - ONENODE – periodic
  - TWONODE – a normal Edge
  - DEGENERATE – single Node  
marks a collapsed  $u$  or  $v$  on a SURFACE  
 $t$  range specifies the limits used for the pcurve (in the Loop)

## Loop – without a reference surface

- ① Free standing connected Edges  
See page 36 for parsing a non-manifold WireBody
  - ② A list of connected Edges associated with a Plane (which does not require pcurves)
- An ordered collection of Edge objects with associated senses that define the connected Loop
  - Segregates space by maintaining material to the left of the running Loop (or traversed right-handed pointing out of the intended volume)
  - No Edges should be Degenerate
  - Member Types: OPEN or CLOSED (comes back on itself)



## Loop – with a reference surface

- ❶ Collections of Edges (like Loops without a surface) followed by a corresponding collection of pcurves that define the  $[u, v]$  trimming on the surface
- Degenerate Edges are required when the  $[u, v]$  mapping collapses like at the apex of a cone (note that the pcurve is needed to be fully defined using the Edge's  $t$  range)
- An Edge may be found in a Loop twice (with opposite senses) and with different pcurves. For example a closed cylindrical surface at the seam – one pcurve would represent the beginning of the period where the other is the end of the periodic range.
- Types: OPEN or CLOSED (comes back on itself)

## Face

- A surface bounded by one or more Loops with associated senses
- Only one outer Loop (sense = SOUTER(1)) and any number of inner Loops (sense = SINNER(-1)). Note that under very rare conditions a Loop may be found in more than 1 Face – in this case the one marked with sense = +/- 2 must be used in a reverse manner.
- All Loops must be CLOSED
- Loop(s) must not contain reference geometry for Planar surfaces
- If the surface is not a Plane then the Loop's reference Object must match that of the Face
- `mtype` is the orientation of the Face based on surface's  $U \otimes V$ :
  - SFORWARD or SREVERSE when the orientations are opposed

Note that this is coupled with the Loop's orientation (i.e. an outer Loop traverses the Face in a right-handed manner defining the outward direction)

## Shell

- A collection of one or more connected Faces that if CLOSED segregates regions of 3-Space
- All Faces must be properly oriented
- Non-manifold Shells can have more than 2 Faces sharing an Edge
- Member Types: OPEN (including non-manifold) or CLOSED
- CLOSED Shells are required for SOLIDBODY Body types

## Body

- Container used to aggregate Topology
- *Owns* all the Objects contained within
- Member Types:
  - WIREBODY – contains a single Loop  
call EG\_getBodyTopos on Edges and compare to the number of Edges in the Loop  
if different → non-manifold therefore use the Edges returned from EG\_getBodyTopos
  - FACEBODY – contains a single Face – IGES import
  - SHEETBODY – contains one or more Shell(s) which can be either non-manifold or manifold (though usually a manifold Body of this type is promoted to a SOLIDBODY)
  - SOLIDBODY:
    - A manifold collection of one or more CLOSED Shells with associated senses
    - There may be only one outer Shell (sense = SOUTER(1)) and any number of inner Shells (sense = SINNER(-1))
    - Edges (except DEGENERATE) found exactly twice (sense =  $\pm 1$ )

<b>Topology</b>	<b>BRep define</b>	<b>Effective define</b>	<b>Note</b>
Model			Container for Bodies, EBodies & Tessellations
Body	BODY	EBODY	An EBody is a modification of a Body w/ “E” entities
Shell	SHELL	ESHELL	1 to 1 mapping
Face	FACE	EFACE	EFace consists of 1 or more Faces
Loop	LOOP	ELOOPX	Collection of EEdges, No pcurves
Edge	EDGE	EEDGE	EEdge consists of 1 or more ordered Edges
Node	NODE	n/a	no ENodes, but not all Nodes in the Body are found in the EBody

## Virtual Topology & BRep closure

- BRep Topology can inhibit *quality* meshes
  - Spurious Nodes
  - Small Edges that could be coalesced
  - Sliver Faces
- EGADS' *Effective Topology*
  - Automatically removes spurious Nodes (unless the Node has .Keep attribute)
  - Automatically coalesces Edges (unless the Edge has .Keep attribute) to make EEdges
  - Collects Faces explicitly or by attribute
    - uses a global  $[u, v]$  mapping driven by an EGADS tessellation
    - Face triangulations must *touch* along at least 1 triangle side
  - EFaces & EEdges contain no geometry, therefore depend on the included BRep Objects
  - Adjusts EFace & EEdge inverse and forward evaluations based on closure at bounds

- *Effective Topology* Objects can maintain their own attributes
- EBody Objects can have their own tessellations.
- *Effective Topology* Objects can NOT be used in construction.
- Nodes that have Degenerate Edges cannot disappear from the EBody. This also means that certain collections of Faces will not be allowed that may remove all Edges supporting that Node.
- EFaces (of more than 1 Face) are always SFORWARD
- The direction of the EEdge is set by the first Edge in the collection. This means that a positive direction is based on the connecting Node to the next Edge, which may have the first Edge traversed in the negative sense.
- The rules for collecting entities in Solid Bodies differ from those of Sheet Bodies where Edges are exposed (only trimming a single Face). In this case an angle criteria is used when making EEdges and removing Nodes.
- A Face may be evaluated throughout the *UVbox* range. This is not true for EFaces containing more than a single Face. An evaluation is only valid based on the original input tessellation.

## Discrete representation of an Object

### Geometry

- Unconnected discretization of the range of the Object
  - Polyline for curves at constant  $t$  increments
  - Regular grid for surfaces at constant increments (isoclines)

### Body/EBody Topology

- Connected and trimmed tessellation including:
  - Polyline for Edges/EEdges
  - Triangulation for Faces/EFaces
  - Optional Quadrilateral Patching for Faces/EFaces
- Ownership and Geometric Parameters for Vertices
- Adjustable parameters for side length and curvature (x2)
- Watertight
- Exposed per Face/Edge or Global indexing



## Control of the use of Quadrilateral Templates

- Automatic with triangulation scheme
- Attempts to isolate 3 or 4 “sides”
  - Only single Loop/ELoop
  - Faces/EFaces with more than 4 Edges/EEdges are analyzed to see if multiple EDGES can be treated as a single “side”
- Point counts on sides (based on Edge/EEdge Tessellation):
  - TFI if opposites are equal
  - Templates otherwise
- Defeated/modified with Body/EBody or Face/EFace attribute `.qParams`
  - If ATTRSTRING – turn off quadding templates
  - If ATTRREAL (3 in length):
    - 1 Edge/EEdge matching expressed as the deviation from alignment [default: 0.05]
    - 2 Maximum quad side ratio point count to allow [default: 3.0]
    - 3 Number of smoothing iterations [default: 0.0]

## Manual Watertight Quadrilateral Face/EFace Treatment

- Requires Existing Body/EBody Tessellation
- Must be able to Isolate 4 “sides”
  - Only single Loops/ELoops
  - Faces/EFaces with more than 4 Edges/EEEdges are analyzed to see if multiple Edges/EEEdges can be treated as a single “side”
  - No DEGENERATE Edges/EEEdges
- Point counts on sides (based on Edge/EEEdge Tessellation):
  - TFI if opposites are equal
  - Templates otherwise
- Can use Edge Tessellation Adjustment Functions when point counts don't allow for Quadding
- See the function `EG_makeQuads` – page 122.

## Body/EBody Watertight Full Quadrilateral Treatment

- Fully automatic & robust
- Method:
  - 1 Starts from an EGADS triangulation of a Body (Tessellation Object)
  - 2 Subdivides all triangle sides so that each triangle becomes 3 quadrilaterals
  - 3 iterate on a regularization scheme:
    - swap/insert/collapse so that the *valence* at each vertex approaches 4 while maintaining a valid tessellation
- Driven by the Edge discretizations, which should start out as double the desired size
- Results in an unstructured tessellation unless the underlying triangulation was derived from TFI
- See the function `EG_quadTess` – page 115.

Models can contain Body, *Effective Topology* and Tessellation Objects, unless they are the output from construction operators. In this case the Model will only contain Body Objects. Models can be created with `EG_makeTopology` (see page 89) and parsed by using `EG_getTopology` (see page 90). These rules apply:

- `nchild` is always the number of Bodies and should be less than or equal to `mtype` (`mtype` can be zero, indicating no ancillary egos).
- You must look at the `oclass` (on the children) of any egos after the Bodies to figure out the kind of object.
- Any Tessellation of EBodies must be listed after the EBody referenced.
- The order of the non-Body children is the same as they were when the Model was created.
- Tessellations and EBodies must be closed when the Model is created.
- Tessellation Objects cannot be reopened once in a Model.
- The Model must contain any Bodies/EBodies referenced by Tessellation Objects and Bodies referenced by EBodies.
- Just like Bodies, Objects in a Model become “owned” by the Model and get deleted when the Model is deleted (and cannot be deleted individually).

## TRANSFORM

- Used when copying Objects to change the root position and orientation

## REFERENCE

- Allows for the management of Objects that refer to other Objects (so that deletion does not invalidate the information)
- An internal Object and is not usually seen by the EGADS programmer

## Parent Child References

- Child Objects used with “make” functions are referenced by the resulting Geometric/Topologic parent Object.

## Object Deletion

- Objects are deleted individual with `EG_deleteObject` (all objects are removed with `EG_close`).
- Referenced child Objects can only be deleted after the parent Object is deleted, i.e. in reverse order of the construction.
- Unconnected Geometric & Topologic Objects (i.e. not in a Body) can be deleted *en masse* by invoking `EG_deleteObject` (see page 52) on the Context.

## BODY

- When created, all child Objects are copied and stored instead of referenced
- The child Objects can be removed with EG\_deleteObject in reverse construction order (or *en masse* with EG\_deleteObject on the Context)

## MODEL

- A Body, EBody or Tessellation Object can be included in only one MODEL (you will get a “reference error” if violated)
- Because Tessellation and EBody Objects reference Body Objects, the Body may need to be copied before creating Tessellations and EBodies if any are to be included in a Model
- Deleting a Model also deletes all child Objects contained within the Model

The following pages provide a reference for the EGADS API. Each block describes the function *signature* first using C/C++, then Fortran, then the **Python call in violet** (when using pyEGADS) and last the **Julia call in green** (when using jlEGADS). For C/C++/Fortran:

- Function names begin with “EG\_” (C/C++)
- Function names begin with “IG\_” for the FORTRAN bindings
- Functions almost always return an integer *error code*
- *Object-based* – procedural, usually with the first argument an **ego**
- Signatures usually have the inputs first, then output argument(s)
- Some outputs may be pointers to lists of *things*
  - EG\_free (page 63) needs to be used when marked as “freeable”
- **El** indicates an EGADSlite, **ET** an *Effective Topology* function, **dot** indicates sensitivity functions exist

See `$ESP_ROOT/include/egads.h` & `egads_dot.h` for a complete listing of the functions.  
See `$ESP_ROOT/include/egadsErrors.h` for a list of the return code **defines**.



The Python EGADS API is built on `ctypes` and mirrors the C/C++ API. Methods have similar names and arguments, which are ordered consistently when possible (optional arguments are placed last).

C-arrays with strides are `lists` of `tuples`.

Import statement: `from pyEGADS import egads`

Main API classes: `egads.Context` `egads.ego` `egads.c_ego`

- `c_ego` is a `ctypes` struct for C function arguments of C `ego` type
- Python classes `Context` and `ego` wrap a `c_ego` and implement EGADS API
  - The wrapped `c_ego` is automatically deleted when a Python class is created from an `egads` method
- Python classes can be created using a `c_ego` generated outside of `egads` using the `egads.c_to_py(c_obj, deleteObject=False)` function
  - `deleteObject` indicates if the `c_ego` should be automatically deleted
- The `c_ego` is retrieved from with `object.py_to_c(takeOwnership=False)`
  - `takeOwnership=True` indicates the returned `c_ego` will no longer be automatically deleted

The Julia EGADS API mirrors the C/C++ API.

Methods have same names and similar arguments (ordered), **except**

- Optional arguments (declared after “;”) use keywords
- Arguments indicating array size are excluded

1D Arrays are indicated as `[]` and 2D Arrays as `[][]`

- Functions with argument `[]([])` take 1DArray (flattened) or 2DArray
- Functions with argument `DataType([])` take single value or 1DArray

Import statement: `import egads`

Usage: `egads.function_name()`

Main API structures: `egads.Ctxt`, `egads.Ego`

- In this document, the prefix `egads.` is omitted

**Context** and **Ego** are **Julia structures** wrapping a C **ego** type

- They can be created using a C **ego** generated outside of **egads** calling  
`egads.Context(obj=ego)`      `egads.Ego(ego,ctxt=Context)`
- The wrapped **ego** is automatically deleted using **Julia finalizers**

Functions modifying input values use signature `! eg., egads.function!()`

Functions **don't** return **icode** status. Rather, an internal error is raised

## Get revision

E1

```
EG_revision(int *imajor, int *iminor, const char **OCCrev);  
call IG_revision(I*4 imajor, I*4 iminor, C** OCCrev)  
imajor, iminor, OCCrev = egads.revision()  
imajor, iminor, OCCrev = revision()
```

**imajor** the returned major revision

**iminor** the returned minor revision number

**OCCrev** the returned revision of OpenCASCADE in use

Returns the version information for both EGADS and OpenCASCADE.

## Open EGADS

E1

```
icode = EG_open(ego *context);  
icode = IG_open(I*8 context)  
context = egads.Context()  
context = Context()
```

**context** the returned Context Object

**icode** the integer return code

Opens and returns a Context object. This is required for the use of all EGADS (except for the above).

## Close a Context

E1

```
icode = EG_close(ego context);  
icode = IG_close(I*8 context)  
    del context
```

**context** the Context Object to close

**icode** the integer return code

Cleans up and closes the Context.

## Delete Object

E1

```
icode = EG_deleteObject(ego object);  
icode = IG_deleteObject(I*8 object)  
    del object
```

**object** the Object to delete

**icode** the integer return code

Deletes an Object (if possible). A positive return indicates that the Object is still referenced by this number of other Objects and has not been removed from the Context. If the Object is the Context then all Geometry/Topology Objects in the Context are deleted except those attached to Body/EBody or Model Objects. You cannot delete lesser *Effective Topology* Objects than an EBody.

E1 Note: Only Objects created in an EGADSlite session may be deleted.

## Read Geometric data from a File

```

icode = EG_loadModel(ego context, int bitFlag, const char *name,
                    ego *model);
icode = IG_loadModel(I*8 context, I*4 bitFlag, C**          name,
                    I*8  model)

model = context.loadModel(name, bitFlag=0)
model = loadModel(context::Context, bitFlag::Int, name::string)

```

**context** the Context Object to receive the geometry

**bitFlag** Options (additive):

- 1 Don't split closed and periodic entities
- 2 Split to maintain at least  $C^1$  in BSPLINES
- 4 Don't maintain Units on STEP/IGES read (always millimeters)
- 8 Try to merge Edges and Faces (with same geometry)
- 16 Load unattached Edges as WireBodies (stp/step & igs/iges)

**name** path of file to load (with extension – case insensitive):

igs/iges	IGES file
stp/step	STEP file
brep	native OpenCASCADE file
egads	native file format with persistent Attributes (splits ignored)

**model** the returned Model Object that was read

**icode** the integer return code

Loads and returns a Model Object from disk and puts it in the Context.

See page 55 for more information on STEP & IGES import/export

## Writes the Model to a File

```
icode = EG_saveModel(const ego object, const char *name);  
icode = IG_saveModel(I*8 object, C** name)  
    object.saveModel(name, overwrite=False)  
    saveModel!(object::Ego, name::string; overwrite::Bool)
```

**object** the Model Object to write

**name** path of file to write, type based on extension (case insensitive):

igs/iges IGES file

stp/step STEP file

brep a native OpenCASCADE file

egads a native file format with persistent Attributes and the ability to write  
EBody and Tessellation data

**icode** the integer return code

Writes the BReps (with optional Tessellation and EBody Objects) contained in the Model to disk. Only writes BRep data for anything but EGADS output. Will not overwrite an existing file of the same name unless `overwrite=True`.

Notes:

- 1 **object** can be a single Body for convenience
- 2 See page 55 for more information on STEP & IGES import/export

## Length Units

EGADS is unitless but the file standards are not. To deal with this the attribute “.lengthUnits” (which is set to a string) is used:

- On import (EG\_loadModel):
  - Any Body loaded has “.lengthUnits” set to the import unit designation
- On export (EG\_saveModel):
  - If “.lengthUnits” is found on the Model Object this unit string is used
  - Otherwise all Bodies are examined for “.lengthUnits” and if any is found it is used
  - If none are found, then “millimeters” is used
  - If there is any inconsistency when examining multiple Bodies – “millimeters” is used

## Attributes

STEP/IGES can only mark Topology with “Name”, “Layer” and “Color”. In EGADS, the “Name” string attribute is handled on all topological objects, “Color” on Faces and Edges, during import/export.

“Color” may be one of “red”, “lred”, “green”, “lgreen”, “blue”, “lblue”, “yellow”, “magenta”, “cyan”, “white”, “black”, or three  $RGB \in [0, 1]$  reals.

Notes:

- STEP supports Body attributes “ColorFace” and “ColorEdge” for setting Body defaults.
- STEP WireBodies do not support Edge “Color” (use Body “ColorEdge”) or Node “Name”
- Edge attributes do not export to IGES when the Edge is associated with two or more Faces.
- IGES does not support “Name” on Nodes

## Update Thread ID in Context

```
icode = EG_updateThread(ego context);  
icode = IG_updateThread(I*8 context)  
context.updateThread()  
updateThread!(context::Context)
```

**context** the Context Object to update

**icode** the integer return code

Resets the Context's owning thread ID to the thread calling this function.

## Make a Transform

```
icode = EG_makeTransform(ego context, double *mat, ego *xform);  
icode = IG_makeTransform(I*8 context, R*8 mat, I*8 xform)  
xform = context.makeTransform(mat)  
xform = makeTransform(context::Context, mat::Float[][])
```

**context** the Context

**mat** the 12 values of the translation/rotation matrix

**xform** the returned Transformation Object

**icode** the integer return code

Makes a Transformation Object from a translation/rotation matrix. The rotation portion [3][3] must be an orthonormal matrix with a single scale.



## Get matrix from Transform Object

```
icode = EG_getTransform(const ego xform, double *mat);  
icode = IG_getTransform(I*8 xform, R*8 mat)  
mat = xform.getTransform()  
mat = getTransform(xform::Ego)
```

**xform** the Transformation Object

**mat** the filled 12 values of the translation/rotation matrix

**icode** the integer return code

Returns the transformation information. This appears like is a column-major matrix that is 4 columns by 3 rows and could be thought of as [3][4] in C/C++ (though is flat) and in Fortran dimensioned as (4, 3).

## Copy and flip the orientation of an Object

```
icode = EG_flipObject(const ego object, ego *newObject);  
icode = IG_flipObject(I*8 object, I*8 newObject)  
newObject = object.flipObject()  
newObject = flipObject(object::Ego)
```

**object** the Object to flip

**newObject** The resultant new Object

**icode** the integer return code

Creates a new EGADS Object by copying and reversing the input object. Can be Geometry (flip the parameterization) or Topology (reverse the sense). Not for Node, Edge, Body or Model. Surfaces reverse only the *u* parameter.

## Copy and optionally Transform an Object

```
icode = EG_copyObject(const ego object, void *other, ego *newObj);  
icode = IG_copyObject(I*8 object, I*8 other, I*8 newObj)  
newObj = object.copy(other, other=None)  
newObj = copyObject(object::Ego; other::Ego)
```

**object** the Object to copy

**other** a Transformation Object, a Body Object, **NULL** for a strict copy, or a vector of **doubles**

**newObj** The resultant new Object

**icode** the integer return code

Creates a new EGADS Object by copying and optionally transforming the input object. A Tessellation or PCurve Object cannot be transformed. For a Tessellation Object, **other** can be a vector of displacements that is 3 times the number of vertices of **doubles** in length to *morph* the tessellation. Also, if **object** is a Tessellation Object or an EBody Object and **other** is a Body Object, the existing Object is copied but associated with the Body specified (not the original referenced object). Note that **other** is not checked if it is compatible with the original referenced Body.

If **other** is a Context, then **object** is copied to this target Context. This is useful in multithreaded settings.

Use `EG_copyGeometry_dot` when requiring sensitivities during construction.

## Copy an Object to the specified Context

```
icode = EG_contextCopy(ego context, const ego object, ego *newObj);  
icode = IG_contextCopy(I*8 context, I*8 object, I*8 newObj)  
newObj = context.contextCopy(object)  
newObj = contextCopy(context::Context; object::Ego)
```

**context** the Context to put the copy (must be a Body for copying a Tessellation Object)

**object** the Object to copy – can be either topology or geometry (but not a PCurve)  
Tessellation Objects may be copied but **context** must be the corresponding Body in the appropriate Context (usually copied before copying the tessellation)  
Note: **object** must not be in the Context

**newObj** The resultant new Object

**icode** the integer return code

This is useful in multithreaded settings when you wish to copy an **object** that exists in a different Context/thread. Use `EG_copyObject` (page 58) when copying from the **object**'s context/thread to the context specified by **other**.

## Get information about an Object

ET, E1

```
icode = EG_getInfo(const ego object, int *oclass, int *mtype,
                  ego *topObj, ego *prev, ego *next);
icode = IG_getInfo(I*8 object, I*4 oclass, I*4 mtype,
                  I*8 topObj, I*8 prev, I*8 next);
oclass, mtype, topObj, prev, next = object.getInfo()
oclass, mtype, topObj, prev, next = getInfo(object::{Ego,Context})
```

**object** the queried Object

**oclass** the returned Object Class

**mtype** the returned Member Type

**topObj** the returned the top level Body/EBody/Model that *owns* **object** or Context

**prev** the returned previous Object in the threaded list (**NULL** at Context)

**next** the returned next Object in the threaded list (**NULL** is the end of the list)

**icode** the integer return code

Queries Object level information. **mtype**'s meanings depend on the returned **oclass** value.

The arguments **topObj**, **prev** and/or **next** can be **NULL** if you do not need these **egos**.

## Get the Context

```
icode = EG_getContext(ego object, ego *context);  
icode = IG_getContext(I*8 object, I*8 *context)  
context = object.context  
context = object.ctx
```

**object** the queried Object

**context** the returned owning Context

**icode** the integer return code

Returns the Context given an object. The context is a property in Python.

## Set the Verbosity Level

E1

```
icode = EG_setOutLevel(ego context, int outLevel);  
icode = IG_setOutLevel(I*8 context, I*4 outLevel)  
context.setOutLevel(outLevel)  
setOutLevel(context::Context, outLevel::Int)
```

**context** the Context

**outLevel** the verbosity level: 0-silent to 3-debug

**icode** the integer return code

Sets the EGADS verbosity level, the default is 1. Success returns the old outLevel.

## Writes a Model to a stream

```
icode = EG_exportModel(ego model, size_t *nbyte, char **stream);
icode = IG_exportModel(I*8 model, I*8 nbyte, CPTR stream)
stream = model.exportModel()
stream = exportModel(model::Ego)
```

**model** the Model Object to export

**nbyte** the returned number of bytes in **stream**

**stream** the returned pointer to the byte-stream (freeable)

Create a stream of data serializing the objects in the Model (including EBodies and Tessellations).

## Loads a Model from a stream

E1 only

```
icode = EG_importModel(ego context, const size_t *nbyte,
                        const char **stream, ego *model);
icode = IG_importModel(I*8 context, I*8 nbyte,
                        CPTR stream, I*8 model)
model = importModel(stream::UInt[])
```

**context** the Context Object to place the import

**nbyte** the number of bytes in **stream**

**stream** the pointer to the byte-stream

**model** the returned Model Object

Deserialize the stream into the objects (Bodies/EBodies/Tessellations) that make up the returned Model.

## Memory Functions

E1

```
EG_free(void *ptr);  
call IG_free(CPTR ptr)
```

```
void *ptr = EG_alloc(size_t nbytes);  
icode = IG_alloc(I*4 nbytes, CPTR ptr)
```

```
void *ptr = EG_calloc(size_t nele, size_t size);  
icode = IG_calloc(I*4 nele, I*4 size, CPTR ptr)
```

```
void *ptr = EG_reall(void *pointer, size_t nbytes);  
icode = IG_reall(CPTR pointer, I*4 nbytes, CPTR ptr)
```

```
char *str = EG_strdup(const char *string);
```

These functions need to be used instead of the C/C++ variants for persistent memory due to the need to allocate/free from the same DLL under Windows.

None of this is necessary within pyEGADS except for one exception – `egads.free(ptr)` releases memory returned from a ctypes interface

## Add an Attribute to an Object

```
icode = EG_attributeAdd(ego object, const char *name, int type,
                        int len, const int *ints, const double *reals,
                        const char *string);
icode = IG_attributeAdd(I*8 object, C**          name, I*4 type,
                        I*4 len, I*4          ints, R*8          reals,
                        C**          string)
object.attributeAdd(name, attrVal)
attributeAdd!(object::Ego, name::String, attrVal::{Number[],String})
```

**object** the Object to attribute  
**name** the name of the attribute  
**type** the attribute type: ATTRINT, ATTRREAL, ATTRSTRING, ATTRCSYS<sup>†</sup> or ATTRPTR  
**len** the number of integers or reals (ignored for strings and pointers)  
**ints** the integers for ATTRINT  
**reals** the floating-point data for ATTRREAL or ATTRCSYS  
**string** the character string pointer for ATTRSTRING or ATTRPTR types  
**icode** the integer return code

Notes: Only the one appropriate **attribute** value (of **ints**, **reals** or **string**) is required.

<sup>†</sup> Use `-attrVal = egads.csystem(reals)` to make a CSYS value.



## Delete an Attribute from an Object

```
icode = EG_attributeDel(ego object, const char *name);  
icode = IG_attributeDel(I*8 object, C**      name)  
    object.attributeDel(name)  
    attributeDel!(object::Ego; name::String)
```

**object** the Object

**name** the name of the attribute to delete

**icode** the integer return code

Deletes an attribute from the Object. If the name is **NULL** (or no argument) then all attributes are removed from this Object.

## The number of Object Attributes

E1

```
icode = EG_attributeNum(ego object, int *nAttr);  
icode = IG_attributeNum(I*8 object, I*4  nAttr)  
nAttr = object.attributeNum()  
nAttr = attributeNum(object::Ego)
```

**object** the Object

**nAttr** the returned number of attributes attached to the Object

**icode** the integer return code

Returns the number of attributes found with this object.

## Return an Attribute on an Object

E1

```

icode = EG_attributeRet(ego object, const char *name, int *type,
                       int *len, const int **ints,
                       const double **reals, const char **string);
icode = IG_attributeRet(I*8 object, C**          name, I*4 type,
                       I*4 len, I*4          ints,
                       R*8          reals, C**          string)

attrVal = object.attributeRet(name)
attrVal = attributeRet(object::Ego, name::String)

```

**object** the Object to query

**name** the name to query

**type** the type: ATTRINT, ATTRREAL, ATTRSTRING, ATTRCSYS or ATTRPTR

**len** the returned number of integers or reals

**ints** the returned pointer to integers for ATTRINT

**reals** the returned pointer to floating-point data for ATTRREAL or ATTRCSYS

**string** the returned pointer to a character string for ATTRSTRING or ATTRPTR types

**icode** the integer return code

Notes: Only the appropriate **attribute** value (of **ints**, **reals** or **string**) is returned.

Care must be taken with the string variable in Fortran not to overstep the declared length.

The CSys (12 reals) is returned in **reals** after the **len** values.

## Get an Attribute on an Object

E1

```

icode = EG_attributeGet(ego object, int index, const char **name,
                        int *type, int *len, const int **ints,
                        const double **reals, const char **string);
icode = IG_attributeGet(I*8 object, I*4 index, C**      name,
                        I*4 type, I*4 len, I*4      ints,
                        R*8      reals, C**      string)

name, attrVal = object.attributeGet(index)
name, attrVal = attributeGet(index::Int)

```

**object** the Object to query

**index** the index (1 to **nAttr** from EG\_attributeNum)

**name** the returned name

**type** the type: ATTRINT, ATTRREAL, ATTRSTRING, ATTRCSYS or ATTRPTR

**len** the returned number of integers or reals

**ints** the returned pointer to integers for ATTRINT

**reals** the returned pointer to floating-point data for ATTRREAL or ATTRCSYS

**string** the returned pointer to a character string for ATTRSTRING or ATTRPTR types

**icode** the integer return code

Notes: Only the appropriate **attribute** value (of **ints**, **reals** or **string**) is returned.

Care must be taken with the string variable in Fortran not to overstep the declared length.

The CSys (12 reals) is returned in **reals** after the **len** values.

## Copy the Attributes from an Object to another

```
icode = EG_attributeDup(ego src, ego dst);  
icode = IG_attributeDup(I*8 src, I*8 dst)  
    dst.attributeDup(src)  
    attributeDup!(src::Ego, dst::Ego)
```

**src** the source Object

**dst** the Object to receive **src**'s attributes

**icode** the integer return code

Deletes an attribute from the destination Object and then copies the source's attributes to the destination. ATTRPTR attributes copy the pointer, other types allocate new data and copy the contents of the source.

## Change Attribute Mode

```
icode = EG_setFullAttrs(ego context, int attrFlag);  
icode = IG_setFullAttrs(I*8 context, I*4 attrFlag)  
    context.setFullAttrs(attrFlag)  
    setFullAttrs!(context::Context, attrFlag::Int)
```

**context** the Context Object

**attrFlag** the mode flag: 0 – the default scheme, 1 – full attribution mode

**icode** the integer return code

Sets the attribution mode for the Context.

- `EG_attributeAdd`'s functionality is the same in both modes. It overwrites an existing attribute with the same name. The name can have the sequence number (when overwriting, not adding). It is an error if the name with sequence does not exist. If the name of the attribute is the “root” name of an existing sequence, this raises an error. Use `EG_attributeAddSeq` to add additional attributes with the same root name to a sequence.
- `EG_attributeDel`, given a name with a sequence number, only deletes that attribute. Note: this will cause resequencing of the attributes such that there is no gap in the attribute sequence numbering. If only one attribute remains in the sequence, the sequence number is removed. Given a name without a sequence number, it will delete all attributes with that root name.
- `EG_attributeNum` returns the number of all attributes including those with sequence numbers.
- `EG_attributeGet` functionality is equivalent in either mode.
- `EG_attributeRet` accepts names with sequence numbers. An error is raised if the input name does not have a sequence number and there are multiple attributes in the sequence. See `EG_attributeRetSeq` on page 72.
- `EG_attributeDup`'s functionality changes depending on the mode. With full attribution it follows the merge rules seen on page 15. That is, if the destination object already has the attributes they will not be overwritten. If you want to overwrite the existing attributes in full mode, invoke `EG_attributeDel` with **NULL** and then call `EG_attributeDup`.

## Add an Attribute to an Object

```
icode = EG_attributeAddSeq(ego object, const char *name, int type,
                           int len, const int *ints,
                           const double *reals, const char *string);
icode = IG_attributeAddSeq(I*8 object, C**      name, I*4 type,
                           I*4 len, I*4      ints,
                           R*8      reals, C**      string)
object.attributeAddSeq(name, attrVal)
attributeAddSeq!(obj::Ego, name::String, attrVal::{Number[],String})
```

- object** the Object to attribute
- name** the name of the attribute
- type** the attribute type: ATTRINT, ATTRREAL, ATTRSTRING, ATTRCSYS<sup>†</sup> or ATTRPTR
- len** the number of integers or reals (ignored for strings and pointers)
- ints** the integers for ATTRINT
- reals** the floating-point data for ATTRREAL or ATTRCSYS
- string** the character string pointer for ATTRSTRING or ATTRPTR types
- icode** the integer return code or sequence number (0, 2 and on)

Notes: Only the appropriate **attribute** value (of **ints**, **reals** or **string**) is required.

<sup>†</sup> Use `-attrVal = egads.csystem(reals)` to make a CSYS value.

If there are no attributes with this name on the object this acts just like `EG_attributeAdd`.

## The number of Sequenced Attributes

El

```
icode = EG_attributeNumSeq(ego object, const char *name, int *nSeq);  
icode = IG_attributeNumSeq(I*8 object, C**          name, I*4  nSeq)  
    nSeq = object.attributeNumSeq(name)  
    nSeq = attributeNumSeq(object::Ego, name::String)
```

**object** the Object

**name** the name of the attribute

**nSeq** the returned number of sequence attributes with the name  
0 for no sequencing, 2 or more for the number of attributes in the root name sequence

**icode** the integer return code

Returns the number of named sequenced attributes found on this object.

## Return a Sequenced Attribute on an Object

E1

```

icode = EG_attributeRetSeq(ego object, const char *name, int index,
                          int *type, int *len, const int **ints,
                          const double **reals, const char **string);
icode = IG_attributeRetSeq(I*8 object, C**          name, I*4 index,
                          I*4 type, I*4 len, I*4          ints,
                          R*8          reals, C**          string)

attrVal = object.attributeRetSeq(name, index)
attrVal = attributeRetSeq(object::Ego, name::String, index::Int)

```

**object** the Object to query

**name** the “root” name to query

**index** the sequence number (1 to **nSeq**)

**type** the type: ATTRINT, ATTRREAL, ATTRSTRING, ATTRCSYS or ATTRPTR

**len** the returned number of integers or reals

**ints** the returned pointer to integers for ATTRINT

**reals** the returned pointer to floating-point data for ATTRREAL or ATTRCSYS

**string** the returned pointer to a character string for ATTRSTRING or ATTRPTR types

**icode** the integer return code

Notes: Only the appropriate **attribute** value (of **ints**, **reals** or **string**) is returned.

Care must be taken with the string variable in Fortran not to overstep the declared length.

The CSys (12 reals) is returned in **reals** after the **len** values.



## Create a Geometry Object

dot

```
icode = EG_makeGeometry(ego context, int oclass, int mtype, ego rGeom,
                        const int *ints, const double *reals,
                        ego *nGeom);
icode = IG_makeGeometry(I*8 context, I*4 oclass, I*4 mtype, I*8 rGeom,
                        I*4      ints, R*8      reals,
                        I*8   nGeom)
nGeom = context.makeGeometry(oclass, mtype, reals, ints=None,
                             geom=None)
nGeom = makeGeometry(ctxt::Ctxt, oclass::Int, mtype::Int, rGeom::Ego,
                     reals::Float[] ([]), ints::Int[])
```

**context** the Context Object

**oclass** the Object Class: PCURVE, CURVE or SURFACE

**mtype** the Member Type (depends on **oclass**)

**rGeom** the reference Geometry Object (if none use **NULL**)

**ints** the integer information (if none use **NULL**)

**reals** the real data used to construct the geometry

**nGeom** the returned pointer to the new Geometry Object

**icode** the integer return code

Notes: **ints** is required for either **mtype** = BEZIER or BSPLINE.

See pages 16-29 for a complete listing of **oclass/mtype** data requirements.

## Query a Geometry Object

dot, El

```

icode = EG_getGeometry(ego object, int *oclass, int *mtype,
                       ego *rGeom, int **ints, double **reals);
icode = IG_getGeometry(I*8 object, I*4 oclass, I*4 mtype,
                       I*8 rGeom, I*4 ints, R*8 reals)

oclass, mtype, reals, ints, rGeom = object.getGeometry()
oclass, mtype, reals, ints, rGeom = getGeometry(object::Ego)

```

**object** the Geometry Object

**oclass** the returned Object Class: PCURVE, CURVE or SURFACE

**mtype** the returned Member Type (depends on **oclass**)

**rGeom** the returned reference Geometry Object (**NULL** if none)

**ints** the returned pointer to integer information (**NULL** if none) (*freeable*)

**reals** the returned pointer to real data used to describe the geometry (*freeable*)

**icode** the integer return code

Notes: **ints** is returned for either **mtype** = BEZIER or BSPLINE.

See pages 16-29 for a complete listing of **oclass/mtype** data information.

## Create a Surface by *skinning* Curves

```
icode = EG_skinning(int nCurve, ego *curves, int degree,  
                   ego *bspline);  
icode = IG_skinning(I*4 nCurve, I*8 curves, I*4 degree,  
                   I*8 bspline)  
bspline = egads.skinning(curves, degree=3)  
bspline = skinning(curves::Ego[], degree::Int)
```

**nCurve** the number of BSpline curves to *skin*

**curves** a pointer to a vector of **egos** containing non-periodic, non-rational BSPLINE curves properly positioned and ordered

**degree** degree of the BSpline used in the *skinning* direction

**bspline** the returned pointer to the new BSpline Surface Object

**icode** the integer return code

This function produces a BSpline Surface that is not fit or approximated in any way, and is true to the input curves.

## Create an Object by Fitting data to a BSpline

dot

```

icode = EG_approximate(ego context, int mDeg, double tol,
                      const int *sizr, const double *xyz, ego *bspl);
icode = IG_approximate(I*8 context, I*4 mDeg, R*8 tol,
                      I*4      sizr, R*8      xyz, I*8  bspl)

bspl = context.approximate(sizr, xyz, mDeg=0, tol=1.e-8)
bspl = approximate(context::Ctxt, sizr::Int[], xyz::Float[][];
                  mDeg::Int, tol::Float)

```

**context** the Context Object

**mDeg**<sup>†</sup> the maximum degree used by OpenCASCADE [3-8], or cubic by EGADS [-3-2]:

-1/0 – fixes the bounds and uses natural end conditions

-2/1 – fixes the bounds and maintains the slope input at the bounds

-3/2 – fixes the bounds & quadratically maintains the slope at 2<sup>nd</sup> order

**tol** is the tolerance to use for the BSpline approximation procedure,  
zero for a SURFACE fit (OpenCASCADE)

**sizr** a vector of 2 integers that specifies the size and dimensionality of the data. If the second is zero, then a CURVE is fit and the first integer is the length of the number of [x, y, z] triads. If the second integer is nonzero, then the input data reflects a 2D map.

**xyz** the data to fit (3 times the number of points in length)

**bspl** the returned pointer to the new BSPLINE Geometry Object

**icode** the integer return code

<sup>†</sup> Notes: negative indicates periodic where the first point must be identical to the last; for SURFACES the periodicity may only be in U; periodic CURVES must be -1 only; this forces equally spaced knots; the result is C2 but not a *periodic* BSPLINE.

## Create an Object by Fitting a BSpline to triangles

```

icode = EG_fitTriangles(ego context, int len, double *xyzs, int ntris,
                        const int *tris, const int *tric, double tol,
                        ego *bspline);

icode = IG_fitTriangles(I*8 context, I*4 len, R*8 xyzs, I*4 ntris,
                        I*4 tris, I*4 tric, R*8 tol,
                        I*8 bspline)

bspline = context.fitTriangles(xyzs, tris, tric=None, tol=1e-7)
bspline = fitTriangles(ctxt::Ctxt, xyzs::Float[]([]), tris::Int[]([]);
                      tric::Int[], tol::Float)

```

**context** the Context Object

**len** the number of vertices in the triangulation

**xyzs** the data to fit (3 times **len** in length)

**ntris** the number of triangles

**tris** the pointer to triangle indices (1 bias) (3 times **ntris** in length)

**tric** the pointer to neighbor triangle indices (1 bias) – 0 or (-) at bounds  
**NULL** – will compute (3 times **ntris** in length, if not **NULL**)

**tol** the is the tolerance to use for the BSpline approximation procedure

**bspline** the returned pointer to the new Geometry Object

**icode** the integer return code

Computes and returns the resultant object created by approximating the triangulation by a BSpline surface.

## Insert additional Knots into a BSpline

```

icode = EG_addKnots(ego bspline,
                    int minDegU, int nU, const double *Us,
                    int minDegV, int nV, const double *Vs, ego *result);
icode = IG_addKnots(I*8 bspline,
                    I*4 minDegU, I*4 nU, R*8 Us,
                    I*4 minDegV, I*4 nV, R*8 Vs, I*8 result)
result = bspline.addKnots(minDegU, Us, minDegV, Vs)
result = addKnots(bspline::Ego, minDegU::Int, Us::Float[],
                  minDegV::Int, Vs::Float[])

```

**bspline** the B-Spline Curve or Surface Object

**minDegU** the minimum U-degree in the resulting B-Spline (Degree for Curve)

**nU** the number of U Knots to insert (Knots for Curve)

**Us** the U-Knot values to insert (may be repeated) (**nU** in length) (Knots for Curve)

**minDegV** the minimum V-degree in the resulting B-Spline (Ignored for Curve)

**nV** the number of V Knots to insert (Ignored for Curve)

**Vs** the v-Knot values to insert (may be repeated) (**nV** in length) (Ignored for Curve)

**result** the returned pointer to the new B-Spline Geometry Object

**icode** the integer return code

The the shape of resulting B-Spline geometry is unchanged.

## Create a CURVE by taking an isocline of a SURFACE

```
icode = EG_isoCline(ego surface, int iUV, double value, ego *curve)
icode = IG_isoCline(I*8 surface, I*4 iUV, R*8 value, I*8 curve)
curve = surface.isoCline(iUV, value)
curve = isoCline(surface::Ego, iUV::Int, value::Float)
```

**surface** the Surface Object

**iUV** the type of isocline: UIISO (0) constant U – or – VISO (1) constant V

**value** the value used for the isocline

**curve** pointer to the returned isocline curve

**icode** the integer return code

Computes from the input surface and returns the isocline curve.

## Check if PCURVE is an isocline

```
icode = EG_isIsoPCure(ego pcurve, int *iUV, double *value, int *fwd)
icode = IG_isIsoPCure(I*8 pcurve, I*4 iUV, R*8 value, I*4 fwd)
isiso, iso, value, fwd = pcurve.isIsoPCure()
isiso, iso, value, fwd = isIsoPCure(pcurve::Ego)
```

**pcurve** the PCurve Object

**iUV** type of isocline (may be NULL): UIISO (0) constant U – or – VISO (1) constant V

**value** the value of the isocline (may be NULL)

**fwd** direction of the PCurve (SFORWARD (1) - Forward, SREVERSE (-1) - Reversed)

**icode** the integer return code (EGADS\_SUCCES if pcurve is an iso, EGADS\_NOTFOUND otherwise)



## Returns the range and periodicity

dot, *ET*, E1

```
icode = EG_getRange(ego object, double *range, int *periodic);
icode = IG_getRange(I*8 object, R*8 range, I*4 periodic)
range, periodic = object.getRange()
range, periodic = getRange(object::Ego)
```

- object** the input Object (PCURVE, CURVE, EDGE, EEDGE, SURFACE, FACE or EFACE)  
**range** PCURVE, CURVE, EDGE, EEDGE – 2 vales are filled:  $t_{start}$  and  $t_{end}$   
 SURFACE, FACE, EFACE – 4 values are filled:  $u_{min}$ ,  $u_{max}$ ,  $v_{min}$  and  $v_{max}$   
**periodic** 0 for non-periodic, 1 for periodic in  $t$  or  $u$ , 2 for periodic in  $v$  (or-able)  
**icode** the integer return code

## Compute the arc-length of an Object

*ET*, E1

```
icode = EG_arcLength(ego object, double t1, double t2, double *alen);
icode = IG_arcLength(I*8 object, R*8 t1, R*8 t2, R*8 alen)
alen = object.arcLength(t1, t2)
alen = arcLength(object::Ego, t1::Float, t2::Float)
```

- object** the input Object (PCURVE, CURVE or EDGE)  
**t1** The starting  $t$  value  
**t2** The end  $t$  value  
**alen** the returned resultant arc-length  
**icode** the integer return code

## Evaluate on the Object

dot, *ET*, E1

```
icode = EG_evaluate(ego object, double *params, double *result);
icode = IG_evaluate(I*8 object, R*8 params, R*8 result)
result = object.evaluate(params)
result = evaluate(object::Ego, params::Float[])
```

**object** the input Object

**params** NODE – ignored (can be **NULL**); PCURVE, CURVE, EDGE, EEDGE – the  $t$  value  
SURFACE, FACE, EFACE –  $u$  then  $v$

**result** the filled returned position, 1<sup>st</sup> and 2<sup>nd</sup> derivatives:

length $\Rightarrow$	Node – 3	PCurve – 6	Edge / EEdge Curve – 9	Face / EFace Surface – 18
Position	$[x, y, z]$	$[u, v]$	$[x, y, z]$	$[x, y, z]$
1 <sup>st</sup>	–	$[du, dv]$	$[dx, dy, dz]$	$[dx_u, dy_u, dz_u]$ $[dx_v, dy_v, dz_v]$
2 <sup>nd</sup>	–	$[du^2, dv^2]$	$[dx^2, dy^2, dz^2]$	$[dx_u^2, dy_u^2, dz_u^2]$ $[dx_{uv}, dy_{uv}, dz_{uv}]$ $[dx_v^2, dy_v^2, dz_v^2]$

**icode** the integer return code – evaluation on EFace may return EGADS\_EXTRAPOL

Note: You cannot evaluate a DEGENERATE Edge/EEdge.

## Inverse evaluation on the Object

*ET, E1*

```

icode = EG_invEvaluate(ego object, double *pos, double *params,
                      double *result);
icode = IG_invEvaluate(I*8 object, R*8      pos, R*8      params,
                      R*8      result)
params, result = object.invEvaluate(pos)
params, result = invEvaluate(object::Ego, pos::Float[])

```

**object** the input Object

**pos** is  $[u, v]$  for a PCURVE and  $[x, y, z]$  for all others

**params** the returned parameter(s) found for the nearest position on the Object:  
 for PCURVE, CURVE, EDGE or EEDGE the one value is  $t$   
 for SURFACE, FACE or EFACE the 2 values are  $u$  then  $v$

**result** the closest position found is returned:  
 $[u, v]$  for a PCURVE (len = 2)  
 $[x, y, z]$  for all others (len = 3)

**icode** the integer return code

Note: When using this with a Face the timing is significantly slower than making the call with the Face's reference surface (due to the clipping). If you don't need this limiting call EG\_invEvaluate with the underlying Surface Object.

## Returns the curvature information on the Object *ET*, E1

```
icode = EG_curvature(ego object, double *params, double *result);
icode = IG_curvature(I*8 object, R*8      params, R*8      result)
result = object.curvature(params)
curvature, direction = curvature(object::Ego, params::Float([]))
```

**object** the input Object

**params** parameter(s) used to compute on the Object:  
PCURVE, CURVE, EDGE, EEDGE – the  $t$  value  
SURFACE, FACE, EFACE –  $u$  then  $v$

**result** the filled returned curvature information:

length $\Rightarrow$	PCurve – 3	Edge / EEdge Curve – 4	Face / EFace Surface – 8
Direction	Curvature [ $dir_u, dir_v$ ]	Curvature [ $dir_x, dir_y, dir_z$ ]	Curvature1 [ $dir1_x, dir1_y, dir1_z$ ]
Direction			Curvature2 [ $dir2_x, dir2_y, dir2_z$ ]

**icode** the integer return code

Note: You cannot get curvature on a DEGENERATE Edge.

## Returns other Curve that matches the input Curve

```

icode = EG_otherCurve(ego object, ego curve, double tol, ego *ocurve);
icode = IG_otherCurve(I*8 object, I*8 curve, R*8 tol, I*8 ocurve)
ocurve = object.otherCurve(curve)
ocurve = otherCurve(object::Ego, curve::Ego; tol::Float)

```

- object** the input Object (SURFACE or FACE)
- curve** the input PCurve or Curve/Edge Object
- tol** is the tolerance to use when fitting the output curve
- ocurve** the returned approximated resultant Curve or PCurve Object
- icode** the integer return code

Produces the PCurve from the Curve/Edge or *vice versa*.

## Do the Objects represent the same Geometry?

```

icode = EG_isSame(const ego object1, const ego object2);
icode = IG_isSame(I*8 object1, I*8 object2)
bool = object1.isSame(object2)
bool = isSame(object1::Ego, object2::Ego)

```

- object1** the first input Object (NODE, CURVE, EDGE, SURFACE or FACE)
- object2** the second input Object (to make the comparison)
- icode** the integer return code (same is EGADS\_SUCCESS, not same is EGADS\_OUTSIDE)

## Concatenates 2 BSpline curves to make another

```
icode = EG_mergeBSplineCurves(ego curve1, ego curve2, ego *bspline);
icode = IG_mergeBSplineCurves(I*8 curve1, I*8 curve2, I*8 bspline)
bspline = curve1.mergeBSplineCurves(curve2)
bspline = mergeBSplineCurves(curve1::Ego, curve2::Ego)
```

**curve1** the first input BSpline CURVE Object

**curve2** the second input BSpline CURVE Object

**bspline** pointer to the returned BSpline CURVE Object

**icode** the integer return code

The starting and ending control points are matched and used to mate the curves. The direction of the result is fixed by **curve1**. The  $t$  range is the sum of the 2 input CURVE ranges.

## Converts geometry to BSPLINE mt type

```
icode = EG_convertToBSpline(ego object, ego *bspline);
icode = IG_convertToBSpline(I*8 object, I*8 bspline)
bspline = object.convertToBSpline()
bspline = convertToBSpline(object::Ego)
```

**object** the input Object (PCURVE, CURVE, EDGE, SURFACE or FACE)

**bspline** pointer to the returned BSpline Geometry Object

**icode** the integer return code

Computes and returns the BSpline representation of the input Geometry Object.

## Converts geometry (with limits) to BSPLINE mt type

```

icode = EG_convertToBSplineRange(ego object, const double *range,
                                ego *bspline);
icode = IG_convertToBSplineRange(I*8 object, R*8 range,
                                I*8 bspline)

bspline = object.convertToBSpline(range)
bspline = convertToBSplineRange(object::Ego, range::Float[]([]))

```

**object** the input Object (PCURVE, CURVE or SURFACE)  
**range**  $t$  range (2) or  $[u, v]$  box (first for  $u$  then for  $v - 4$ ) to limit the conversion  
**bspline** pointer to the returned BSpline Geometry Object  
**icode** the integer return code

Required when converting Geometry Objects with infinite range.

oclass	mtype	Notes
MODEL	→	Number of total <b>egos</b> or zero see page 44
BODY	WIREBODY FACEBODY SHEETBODY  SOLIDBODY	A single Loop A single Face A single non-manifold or manifold Shell A <i>Solid</i>
SHELL	OPEN or CLOSED	
FACE	SREVERSE or SFORWARD	orientation of surface vs. Face
LOOP	OPEN or CLOSED	
EDGE	DEGENERATE  ONENODE TWNODE	a single Node marking the collapse of a surface (nchild = 1) a CLOSED curve (nchild = 1) a normal Edge (nchild = 2)
NODE	–	



## Create a Topology Object

dot

```

icode = EG_makeTopology(ego context, ego geom, int oclass, int mtype,
                        double *reals, int nchild, ego *children,
                        int *senses, ego *topo);
icode = IG_makeTopology(I*8 context, I*8 geom, I*4 oclass, I*4 mtype,
                        R*8      reals, I*4 nchild, I*8  children,
                        I*4  senses, I*8  topo)

topo = context.makeTopology(oclass, mtype=0, geom=None, reals=None,
                           children=None, senses=None)

topo = makeTopology!(ctxt::Context, oclass::Int; mtype::Int,
                    geom::Ego, reals::Float[], senses::Int, children::Float([]))

```

**context** the Context Object

**geom** the reference Geometry Object (if none use **NULL**)

**oclass** the Object Class: NODE, EDGE, LOOP, FACE, SHELL, BODY or MODEL

**mtype** the Member Type (depends on **oclass** – see page 88)

**reals** the real data: may be **NULL** except for NODE that contains the  $[x, y, z]$  location and EDGE where the  $t_{min}$  and  $t_{max}$  (the parametric bounds) are specified

**nchild** number of children (lesser) Topological Objects

**children** vector of children objects (**nchild** in length)

if a LOOP with a reference SURFACE, then  $2 * \text{nchild}$  in length (PCurves follow)

**senses** a vector of children integer senses: SFORWARD/SREVERSE for LOOP, and SOUTER/SINNER for FACE **nchild** > 1 (may be **NULL** for FACE **nchild** = 1)

**topo** the returned pointer to the new Topology Object

## Query a Topology Object 1/2

ET, E1

```

icode = EG_getTopology(ego topo, ego *geom, int *oclass, int *mtype,
                      double *reals, int *nchild, ego **children,
                      int **senses);

icode = IG_getTopology(I*8 topo, I*8 geom, I*4 oclass, I*4 mtype,
                      R*8 reals, I*4 nchild, I*8 children,
                      I*4 senses)

oclass, mtype, geom, reals, children, senses = topo.getTopology()
oclass, mtype, geom, reals, children, senses = getTopology(topo::Ego)

```

- topo** the Topology or *Effective Topology* Object to query
- geom** the returned reference Geometry Object (can be **NULL**)
- oclass** the returned Object Class: Topology or *Effective Topology*
- mtype** the returned Member Type (depends on **oclass** – see page 88)
- reals** the real data (at most 4 **double**s are filled): NODE – contains the  $[x, y, z]$  location, EDGE where the  $t_{min}$  and  $t_{max}$  (the parametric bounds) are returned and FACE where the  $[u, v]$  box is filled → the limits first for  $u$  then for  $v$  (4 in length)
- nchild** the returned number of children (lesser) Topological Objects
- children** the returned pointer to a vector of children objects (**nchild** in length)
  - if a LOOP with a reference SURFACE, then  $2 \times \text{nchild}$  in length (PCurves follow)
  - if a MODEL – **nchild** is the number of Body Objects, **mtype** the total **ego** count
- senses** a vector of senses for the children (LOOPS) or inner/outer for (FACES & SHELLS)
- icode** the integer return code (EGADS\_OUTSIDE for an open EBody)

## Query a Topology Object 2/2

ET

Returns using *Effective Topology*:

EBODY	<b>geom</b> is the source Body Object <b>children</b> are ESHELLs or EFACE (for <b>mtype</b> is FACEBODY) returns EGADS_OUTSIDE if the EBody is still open
ESHELL	<b>children</b> are EFaces
EFACE	no reference geometry, returns $[u, v]$ box for Face collection <b>children</b> are ELoops
ELOOPX	<b>children</b> are EEdges
EEDGE	no ref geometry, returns $t$ range for the Edge collection <b>children</b> are Nodes

Use EG\_getBodyTopos (page 96) on the EBody with **ref** as an EFace and **oclass** as FACE to find all of the Faces in that EFace. The same can be done with Edges/EEdges or use the function EG\_effectiveEdgeList (page 155).

## Create a Face Object

```
icode = EG_makeFace(ego object, int mtype, const double *rdata,
                    ego *face);
icode = IG_makeFace(I*8 object, I*4 mtype, R*8 rdata,
                    I*8 face)
face = object.makeFace(mtype, rdata=None)
face = makeFace(object::Ego, mtype::Int; rdata::Float[]([]))
```

- object** either a Loop (for a planar cap), a surface with  $[u, v]$  bounds, or a Face to be hollowed out
- mtype** is either SFORWARD or SREVERSE. SFORWARD gives a Face normal vector consistent with orientation of the Loop or the normal vector of a surface. Ignored when the input object is a Face
- rdata** may be **NULL** for Loops, but must be the limits for a surface (4 values), the hollow/offset distance and fillet radius (zero is for no fillets) for a Face input object (2 values)
- face** the resultant returned topological Face Object (a return of EGADS\_OUTSIDE is the indication that offset distance was too large to produce any cutouts, and this result is the input **object**)
- icode** the integer return code

This *helper* function creates a simple Face from a Loop or a surface. Also can be used to hollow out a Face with a single Loop. This function creates any required Nodes, Edges and Loops.

## Create a Loop Object

```
icode = EG_makeLoop(int nedge, ego *edges, ego geom, double toler,
                    ego *loop);
icode = IG_makeLoop(I*4 nedge, I*8 edges, I*8 geom, R*8 toler,
                    I*8 loop)
loop, edges = egads.makeLoop(edges, geom=None, toler=0.0)
loop, edges = makeLoop!(edges::Ego{[]}; geom::Ego, toler::Float)
```

**nedge** the number of Edge Objects in the list ( $\geq 1$ )

**edges** list of Edge Objects, of which some may be **NULL** (**nedge** in length)

Note: list entries are **NULL**ified when included in Loops

**geom** the Surface Object for non-planar Loops to be used to bound Faces (can be **NULL**)

**toler** tolerance used for the operation (0.0 – use the Edge tolerances)

**loop** the resultant Loop Object

**icode** the integer return code or the number of non-**NULL** entries in **edges** when returned

This *helper* function creates a Loop Object from a list of Edge Objects, where the Edges (not DEGENERATE) do not have to be topologically connected. The tolerance is used to build the Nodes for the Loop. The orientation is set by the first non-NULL entry in the list, which is taken in the positive sense (if closed – otherwise the orientation is not fixed). This is designed to be executed until all list entries are exhausted.

## Create a simple Solid Body

```
icode = EG_makeSolidBody(ego context, int stype, const double *data,
                        ego *body);
icode = IG_makeSolidBody(I*8 context, I*4 stype, R*8 data,
                        I*8 body)

body = context.makeSolidBody(stype, data)
body = makeSolidBody(context::Ctxt, stype::Int, data::Float[])
```

**context** the Context Object

**stype** one of: BOX, SPHERE, CONE, CYLINDER, TORUS

**data** length and fill depends on **stype**:

BOX	6	$[x, y, z]$ then $[dx, dy, dz]$ for the size of box
SPHERE	4	$[x, y, z]$ of center then the radius
CONE	7	apex $[x, y, z]$ , base center $[x, y, z]$ , then the radius
CYLINDER	7	2 axis points and the radius
TORUS	8	$[x, y, z]$ of center, direction of rotation, then the major radius and minor radius

**body** the resultant Solid Body Object

**icode** the integer return code

## Creates a non-manifold Wire Body

```
icode = EG_makeNmWireBody(int nObject, ego *objects, double toler,  
                           ego *wbody);  
icode = IG_makeNmWireBody(I*4 nObject, I*8 objects, R*8 toler,  
                           I*8 wbody)  
wbody = egads.makeNmWireBody(objects, toler=0.0)  
wbody = makeNmWireBody(objects::Ego[]; toler::Float)
```

- nObject** the number of Edge Objects in the list **objects**  
**objects** a pointer to a vector of Edge Objects to make the Wire Body (**nObject** in length)  
**toler** Node tolerance to connect Edges (0.0 indicates the use of the Nodes directly)  
**wbody** the returned pointer to a Wire Body Object  
**icode** the integer return code

### Notes:

1. The collection of Edges in **objects** must be connected in a non-manifold manner.
2. If the Edges reflect a simple (manifold) Loop, use `EG_makeTopology` to make the Loop (so that the senses can be set) and then also construct the Wire Body from the Loop.
3. Edges must be given in an order such that every Edge (after the first) creates no more than one new Node.

## Queries the Objects in a Body

ET, E1

```

icode = EG_getBodyTopos(const ego body, ego ref, int oclass,
                        int *ntopo, ego **topos);
icode = IG_getBodyTopos(I*8 body, I*8 ref, I*4 oclass,
                        I*4 ntopo, CPTR topos)
topos = object.getBodyTopos(oclass, ref=None)
topos = getBodyTopos(body::Ego, oclass::Int; ref::Ego)

```

**body** the Body/EBody Object

**ref** reference Topology Object or **NULL**. Sets the context for the returned Objects (i.e., all objects of the class **oclass** in the tree looking towards that class from **ref**)  
**NULL** starts from the **body** (for example all Nodes in the Body)

**oclass** is NODE, EDGE, LOOP, FACE or SHELL – must not be the same class as **ref**  
 for EBODY can be EEDGE, ELOOPX, EFACE, ESHELL or the above

**ntopo** the returned number of Topology Objects

**topos** is a returned pointer to the vector of Objects, it is possible that an individual Object may be **NULL** (*freeable*)

Note: the argument can be **NULL** so the Objects are not filled

**icode** the integer return code

This allows for the traversal of the Topology *tree* by jumping levels and/or looking up the hierarchy.



## Get the index of the Object in a Body

ET, E1

```
index = EG_indexBodyTopo(const ego body, const ego obj);  
index = IG_indexBodyTopo(I*8 body, I*8 obj)  
index = body.indexBodyTopo(obj)  
index = indexBodyTopo(body::Ego, obj::Ego)
```

**body** the Body/EBody Object

**obj** is the Topology Object in the Body/EBody

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

## Get the Object in a Body by index

ET, E1

```
icode = EG_objectBodyTopo(const ego body, int oclass, int index,  
                           ego *obj);  
icode = IG_objectBodyTopo(I*8 body, I*4 oclass, I*4 index,  
                           I*8 obj)  
obj = body.objectBodyTopo(oclass, index)  
obj = objectBodyTopo(body::Ego, oclass::Int, index::Int)
```

**body** the Body/EBody Object

**oclass** the Topology/*Effective Topology* object class

**index** the index (bias 1) of the entity requested

**obj** is the returned Topology Object from the Body/EBody

**icode** the integer return code

## Compute the Area

ET

```

icode = EG_getArea(ego object, const double *limits, double *area);
icode = IG_getArea(I*8 object, R*8 limits, R*8 area)
area = object.getArea(limits=None)
area = getArea(object::Ego; limits::Float[]([]))

```

**object** either a Loop (for a planar cap), a surface with  $[u, v]$  bounds or a Face

**limits** may be **NULL** except must contain the limits for a surface (4 words)

**area** the resultant surface area returned

**icode** the integer return code

Computes the surface area from a Loop, a surface or a Face. The resultant area is negative if a non-planar Loop is left-handed relative to the normal of the associated surface.

## Return the Bounding Box info

ET, E1

```

icode = EG_getBoundingBox(const ego object, double *bbox);
icode = IG_getBoundingBox(I*8 object, R*8 bbox)
bbox = object.getBoundingBox()
bbox = getBoundingBox(object::Ego)

```

**object** any topological object

**bbox** 6 **doubles** filled reflecting  $[x, y, z]_{min}$  and  $[x, y, z]_{max}$

**icode** the integer return code

Computes the smallest Cartesian bounding box surrounding the **object**.

## Returns the Mass Properties

ET

```

icode = EG_getMassProperties(const ego object, double *props);
icode = IG_getMassProperties(I*8 object, R*8 props)
volume, areaOrLen, CG, I = object.getMassProperties()
volume, areaOrLen, CG, I = getMassProperties(object::Ego)

```

**object** can be EDGE, LOOP, FACE, SHELL, BODY or *Effective Topology* counterpart  
**props** 14 **doubles** filled reflecting Volume, Area (or Length), Center of Gravity (3) and the inertia matrix at CG (9)  
**icode** the integer return code

Computes and returns the physical and inertial properties of a Topology Object.  
 See EG\_tessMassProperties (page 135) if you want derivatives.

## Do the two Objects represent the same Topology?

```

icode = EG_isEquivalent(const ego topo1, const ego topo2);
icode = IG_isEquivalent(I*8 topo1, I*8 topo2)
bool = topo1.isEquivalent(topo2)
bool = isEquivalent(topo1::Ego, topo2::Ego)

```

**topo1** the first input Topology Object  
**topo2** the second input Topology Object (to make the comparison)  
**icode** the integer return code (same as EGADS\_SUCCESS, not equal is EGADS\_OUTSIDE)

Compares two topological objects for equivalence.

## Inside Predicate

E1

```
icode = EG_inTopology(const ego object, const double *xyz);  
icode = IG_inTopology(I*8 object, R*8 xyz)  
bool = object.inTopology(xyz)  
bool = inTopology(object::Ego, xyz::Float[])
```

**object** the object, can be EDGE, FACE, SHELL or SOLIDBODY

**xyz** the coordinate location to check (3 in length)

**icode** the integer return code (in is EGADS\_SUCCESS, out is EGADS\_OUTSIDE)

Computes whether the point is on or contained within **object**. Works with Edges and Faces by projection. Shells must be CLOSED.

## In Face Predicate

ET, E1

```
icode = EG_inFace(const ego face, const double *uv);  
icode = IG_inFace(I*8 face, R*8 uv)  
bool = face.inFace(uv)  
bool = inFace(face::Ego, uv::Float[])
```

**face** the Face Object

**uv** the  $[u, v]$  location to check (2 in length)

**icode** the integer return code (in is EGADS\_SUCCESS, out is EGADS\_OUTSIDE)

Queries whether the  $[u, v]$  location in the valid part of the Face Object.

## Get Edge's UV on Face

ET, E1

```

icode = EG_getEdgeUV(const ego face, const ego edge, int sense,
                    double t, double *uv);
icode = IG_getEdgeUV(I*8 face, I*8 edge, I*4 sense,
                    I*4 t, R*8 uv)
uv = face.getEdgeUV(edge, t, sense=0)
uv = getEdgeUV(face::Ego, edge::Ego, sense::Int, t::Float)

```

**face** the Face/EFace Object

**edge** the Edge/EEdge Object

**sense** can be 0, but must be specified ( $\pm 1$ ) if **edge** is found in **face** twice, which denotes the position in the Loop to use.

**t** the Edge parametric  $t$  value

**uv** the resultant  $[u, v]$  – 2 values filled.

**icode** the integer return code – EGADS\_TOPOERR when **sense** is 0 with a double Edge hit.

## Notes:

1. Evaluates the Edge/pcurve to get  $[u, v]$  on the Face. Use instead of EG\_invEvaluate (page 83) at the Face/EFace Object's bounds.
2. Cannot be used on an EFace containing more than one Face and a DEGENERATE EEdge.

## Sews unconnected Faces together

```
icode = EG_sewFaces(int nObject, ego *objects, double toler, int flag,
                    ego *model);
icode = IG_sewFaces(I*4 nObject, I*8 objects, R*8 toler, I*4 flag,
                    I*8 model)
model = egads.sewFaces(objects, toler=0.0, manifold=True)
model = sewFaces(objects::Ego[]; toler::Float, flag::Bool)
```

**nObject** the number of Objects in the list **objects**

**objects** a pointer to a vector of **egos** to sew together (**nObject** in length)  
can be Faces, Shells and/or Body Objects (but not WIREBODY)

**toler** tolerance used for the operation (0.0 – use Face tolerances)

**flag** 0 - manifold, 1 - allow for non-manifold results

**model** the returned pointer to a Model Object

**icode** the integer return code

Creates a Model from a collection of Objects by *sewing* Edges that are closer together than the tolerance. The input list can contain Body (not WIREBODY), Shell and/or Face Objects. After the sewing operation, any loose Faces are made into FaceBody Objects, any open Shells made into SheetBody Objects, closed Shells become SolidBody Objects and all are returned in the Model.

## Replace Faces in a Body

```
icode = EG_replaceFaces(const ego body, int nFace, ego *objects,
                        ego *result);
icode = IG_replaceFaces(I*8 body, I*4 nFace, I*8 objects,
                        I*8 result)
result = body.replaceFaces(objects)
result = replaceFaces(body::Ego, objects::Ego[])
```

**body** the Body Object to adjust (either a SHEETBODY or a SOLIDBODY)

**nFace** the number of Face pair Objects in the list **objects**

**objects** a pointer to a vector of **egos** to sew together ( $2*nFace$  in length)  
where the first in the pair must be a Face in **body** and second is either the Face to use  
as a replacement or a **NULL** which indicates that the Face is to be removed

**result** the resultant Body Object, either a SHEETBODY or a SOLIDBODY (where the input  
was a SOLIDBODY and all Faces are replaced in a way that the Loops match up  
– or – if all inner Shell Faces are removed)

**icode** the integer return code

Creates a new SheetBody or SolidBody Object from an input Body and a list of Faces to modify. The Faces are input in pairs where the first must be an Object in the Body and the second either a new Face or **NULL**. The **NULL** replacement flags removal of the Face.

## Removes Nodes from a Body

```
icode = EG_removeNodes(const ego body, int nNode, ego *nodes,
                        ego *result);
icode = IG_removeNodes(I*8 body, I*4 nNode, I*8 nodes,
                        I*8 result)
result = body.removeNodes(nodes)
result = removeNodes(body::Ego, nodes::Ego[])
```

- body** the Body Object to adjust (either a FACEBODY, SHEETBODY or a SOLIDBODY)
- nNode** the number of Node Objects in the list **nodes**
- nodes** a pointer to a vector of NODE **egos** (**nNode** in length)
- result** the resultant Body Object, either a FACEBODY, SHEETBODY or a SOLIDBODY depending on the type of **body**
- icode** the integer return code

Creates a new Body (with the same type as input) from a Body Object and a list of Nodes to remove. The Nodes must only touch 2 Edges and these Edges must bound the same 2 Faces (a SHEETBODY or FACEBODY can bound one Face when on Edges without 2 Face neighbors). If the Edge Curves are not BSplines they are converted and concatenated into a single BSpline Curve and the 2 Edges are replaced by one. Note: you cannot remove a Node that will result in a closed Edge.



## Retrieve the Body Object

*ET, E1*

```
icode = EG_getBody(const ego object, ego *body);  
icode = IG_getBody(I*8 object, I*8 body)  
body = object.getBody()  
body = getBody(body::Ego)
```

**object** the input topology Object (can be an EBody to retrieve the referenced Body)

**body** the returned Body/EBody Object (may be **NULL** if **object** is not attached to a Body).  
An Effective **object** returns an EBody.

**icode** the integer return code

## Return the Tolerance

*E1*

```
icode = EG_getTolerance(const ego topo, double *toler);  
icode = IG_getTolerance(I*8 topo, R*8 toler)  
toler = topo.getTolerance()  
toler = getTolerance(topo::Ego)
```

**topo** the Topology Object

**toler** the returned tolerance (Effective Topology inputs will return 0.0)

**icode** the integer return code

Returns the internal tolerance defined for the Object.

## Return the Maximum Tolerance

*ET, E1*

```
icode = EG_tolerance(const ego topo, double *toler);  
icode = IG_tolerance(I*8      topo, R*8      toler)  
toler = topo.tolerance()  
toler = tolerance(topo::Ego)  
    topo  the Topology Object  
    toler the returned tolerance (Effective Topology returns discrete displacement)  
    icode the integer return code
```

Returns the maximum tolerance defined for the Object's hierarchy.

## Sets the Tolerance – Deprecated

```
icode = EG_setTolerance(const ego topo, double toler);  
icode = IG_setTolerance(I*8      topo, R*8      toler)  
    topo  the Topology Object  
    toler the specified tolerance  
    icode the integer return code
```

Sets the internal tolerance defined for the object. Was useful for SBOs – now use `EG_generalBoolean` (see page 138).

## Find matched Edges between Body Objects

```
icode = EG_matchBodyEdges(const ego body1, const ego body2,  
                           double tolScale, int *nMatch, int **matches)  
icode = IG_matchBodyEdges(I*8 body1, I*8 body2,  
                           R*8 tolScale, I*4 nMatch, CPTR matches)  
matches = body1.matchBodyEdges(body2, tolScale=0.0)  
matches = matchBodyEdges(body1::Ego, body2::Ego; tolScale::Float)
```

**body1** the first input Body Object

**body2** the second Body Object

**tolScale** tolerance scale factor, tolScale > 1 relaxes and 1 > tolScale > 0 tightens tolerance (may be 0.0 for default tolerances)

**nMatch** the number of matched Edge pairs in the list

**matches** pointer to a list of Edge pairs, returned as **NULL** if **nMatch** is zero, otherwise it is a pointer to 2\***nMatch** integers, where each pair are the matching 1-bias indices in the respective bodies (freeable)

**icode** the integer return code

Examines the Edges in one Body against all of the Edges in another. If the number of Nodes, the Node locations, the Edge bounding boxes and the Edge arc-lengths match it is assumed that the Edges match. A list of pairs of 1-bias indices is returned.

## Find matched Faces between Body Objects

```

icode = EG_matchBodyFaces(const ego body1, const ego body2,
                           double tolScale, int *nMatch, int **matches)
icode = IG_matchBodyFaces(I*8 body1, I*8 body2,
                           R*8 tolScale, I*4 nMatch, CPTR matches)
matches = body1.matchBodyFaces(body2, tolScale=0.0)
matches = matchBodyFaces(body1::Ego, body2::Ego; tolScale::Float)

```

**body1** the first input Body Object

**body2** the second Body Object

**tolScale** tolerance scale factor,  $\text{tolScale} > 1$  relaxes and  $1 > \text{tolScale} > 0$  tightens tolerance (may be 0.0 for default tolerances)

**nMatch** the number of matched Face pairs in the list

**matches** pointer to a list of Face pairs, returned as **NULL** if **nMatch** is zero, otherwise it is a pointer to  $2 * \text{nMatch}$  integers, where each pair are the matching 1-bias indices in the respective bodies (freeable)

**icode** the integer return code

Examines the Faces in one Body against all of the Faces in another. If the number of Loops, number of Nodes, the Node locations, number of Edges, the Face and Edge bounding boxes and the Edge arc-lengths match it is assumed that the Faces match. A list of pairs of 1-bias indices is returned.

Note: This is useful for the situation where there are glancing Faces and a UNION operation fails (or would fail). Simply find the matching Faces and do not include them in a call to `EG_sewFaces`, see page 102.

## Maps the Objects of the Source to the Destination

```
icode = EG_mapBody(const ego src, const ego dst,  
                  const char *fAttr, ego *mapped);  
icode = IG_mapBody(I*8      src, I*8      dst,  
                  C**      fAttr, I*8     mapped)  
mapped = src.mapBody(dst, fAttr)  
mapped = mapBody(src::Ego, dst::Ego, fAttr::String)
```

**src** the source Body Object (not a WIREBODY)

**dst** the destination Body Object

**fAttr** the Face attribute used to map the Faces

**mapped** the mapped resultant Body Object copied from **dst**

If **NULL** and **icode** is EGADS\_SUCCESS, **dst** is equivalent and can be used directly in EG\_mapTessBody – see page 117.

**icode** the integer return code

Checks for topological equivalence between **src** and **dst**. If necessary, produces a mapping (indices in **src** which map to **dst**) and places these as body attributes on **mapped** (named “.nMap”, “.eMap” and “.fMap”). Also may modify BSplines associated with Faces.

Note: It is the responsibility of the caller to have uniquely attributed all Face Objects in both **src** and **dst** to aid in the mapping for all but FACEBODYs.

## Maps the Objects of the Source to the Destination

```
icode = EG_mapBody2(const ego src, const char *fAttr,  
                    const char *eAttr, ego dst);  
icode = IG_mapBody2(I*8 src, C** fAttr,  
                    C** eAttr, I*8 dst)  
src.mapBody2(fAttr, eAttr, dst)  
mapBody2(src::Ego, fAttr::String, eAttr::String, dst::Ego)
```

**src** the source Body Object

**dst** the destination Body Object

**fAttr** the Face attribute used to map the Faces

**eAttr** the Edge attribute used to map the Edges

**icode** the integer return code

Checks for topological equivalence between **src** and **dst**. If necessary, produces a mapping (indices in **src** which map to **dst**) and places these as body attributes on **dst** (named “.nMap”, “.eMap” and “.fMap”). Unlike EG\_mapBody, EG\_mapBody2 also works on FACEBODYS and WIREBODYS.

Note: It is the responsibility of the caller to have uniquely attributed all Face and non degenerate Edge Objects in both **src** and **dst** to aid in the mapping.

## Computes the Winding Angle along an Edge

E1

```
icode = EG_getWindingAngle(ego edge, double t, double *angle);  
icode = IG_getWindingAngle(I*8 edge, R*8 t, R*8 angle)  
angle = edge.getWindingAngle(t)  
angle = getWindingAngle(edge::Ego, t::Float)
```

**edge** the input Edge (not DEGENERATE)

returns an error if there are 3 or more Faces attached to the Edge

**t** the  $t$  value along the Edge used to compute the Winding Angle

**angle** the returned Winding Angle in degrees (0.0 – 360.0)

This is measured from one Face “winding” around to the other based on the normals

An Edge with a single Face always returns 180.0

**icode** the integer return code

## Creates a Discrete Object from Geometry

```

icode = EG_makeTessGeom(ego geom, double *limits, int *sizes,
                        ego *tess);
icode = IG_makeTessGeom(I*8 geom, R*8 limits, I*4 sizes,
                        I*8 tess)

tess = geom.makeTessGeom(limits, sizes)
tess = makeTessGeom(geom::Ego, limits::Float[], sizes::Int[])

```

**geom** the input Object, may be a CURVE or SURFACE

**limits** the bounds of the tessellation (like input like the *range* – see EG\_getRange, page 81)

**sizes** a set of 2 integers that specifies the size and dimensionality of the data. The second is assumed zero for a CURVE and in this case the first integer is the length of the number of evenly spaced (in  $t$ ) points created. The second integer must be nonzero for SURFACES and this then specifies the density of the  $[u, v]$  map of coordinates produced (again evenly spaced in the parametric sense). If a value of sizes is negative, then the fill is reversed for that coordinate.

**tess** the returned resultant Tessellation of the geometric entity

**icode** the integer return code

Creates a discretization object from a geometry-based Object.



## Returns the Discrete Object data

```
icode = EG_getTessGeom(ego tess, int *sizes, double **xyzs);  
icode = IG_getTessGeom(I*8 tess, I*4 sizes, CPTR xyzs)  
sizes, xyzs = tess.getTessGeom()  
sizes, xyzs = getTessGeom(tess::Ego)
```

- tess** the input geometric Tessellation Object
- sizes** a returned (filled) set of 2 integers that specifies the size and dimensionality of the data. If the second is zero, then it is from a CURVE and the first integer is the length of the number of  $[x, y, z]$  triads. If the second integer is nonzero then the input data reflects a 2D map of coordinates.
- xyzs** the returned pointer to the suite of coordinate data
- icode** the integer return code

Retrieves the data associated with the discretization of a geometry-based Object.

## Creates a Discrete Object from a Body

ET, E1

```

icode = EG_makeTessBody(ego body, double *parms, ego *tess);
icode = IG_makeTessBody(I*8 body, R*8      parms, I*8  tess)
  tess = body.makeTessBody(parms)
  tess = makeTessBody(body::Ego, parms::Float[])

```

- body** the input Body or closed EBody Object, may be any Body type
- parms** a set of 3 parameters that drive the Edge discretization and the Face triangulation. The first is the maximum length of an Edge segment or triangle side (in physical space); a zero is no limit, and a negative value only tessellates Edges. The second is a curvature-based value that looks locally at the deviation between the centroid of the discrete object and the underlying geometry. Any deviation larger than the input value will cause the tessellation to be enhanced in those regions. The third is the maximum interior dihedral angle (in degrees) between triangle facets (or Edge segment tangents for a WIREBODY tessellation), note that a zero ignores this phase.
- tess** the returned resultant Tessellation of **body**
- icode** the integer return code

## Notes:

1. the attribute “.tParams” on the Body, Faces or Edges overrides **parms** locally (the minimum is used). this attribute must be ATTRREAL, have 3 values (as described above).
2. the ATTRREAL attribute “.tPos” in an Edge directly sets the *ts* for interior Edge positions.
3. the ATTRREAL attribute “.rPos” sets the relative spacing (in arc-length) for interior Edge positions.
4. the ATTRINT attribute “.nPos” sets the relative spacing (evenly in arc-length) for interior Edge positions.
5. an ATTRINT attribute “.tPos” or “.rPos” of length 1 and containing a zero indicates no interior points.

## Convert a Triangulation to Quads

*ET, E1*

```
icode = EG_quadTess(ego tess, ego *qtess);  
icode = IG_quadTess(I*8 tess, I*8 qtess)  
qtess = tess.quadTess()  
qtess = quadTess(tess::Ego)
```

**tess** the input Tessellation Object (not for WIREBODY)  
**qtess** the returned fully quadrilateral Tessellation Object  
**icode** the integer return code

Takes a triangulation as input and outputs a full quadrilateralization of the Body. The algorithm uses the bounds of each Face (the discretization of the Loops) and drives the interior towards regularization (4 quad sides attached to a vertex) without regard to spacing but maintaining a valid mesh. This is the recommended quad approach in that it is robust and does not require manual intervention like EG\_makeQuads – page 122 (plus retrieving the quads is much simpler and does not require invoking EG\_getQuads – page 124 and EG\_getPatch – page 125).

**qtess** is a triangle-based Tessellation Object, but with pairs of triangles (as retrieved by calls to EG\_getTessFace – page 120) representing each quadrilateral. This is marked by the following attributes on **qtess**:

- “.tessType” (ATTRSTRING) is set to “Quad”
- “.mixed” with type ATTRINT and the length is the number of Faces in the Body, where the values are the number of quads (triangle pairs) per Face. Single triangles are followed by triangle pairs for a Face with both triangle and quads.

## Redoes parts of the tessellation for a Body

*ET, E1*

```
icode = EG_remakeTess(ego tess, int nobj, ego *facedg, double *parms);  
icode = IG_remakeTess(I*8 tess, I*4 nobj, I*8 facedg, R*8 parms)  
    tess.remakeTess(facedg, parms)  
    remakeTess!(tess::Ego, facedg::Ego[], parms::Float[])
```

- tess** the Tessellation Object to modify
- nobj** number of Objects in the Face/Edge list
- facedg** a pointer to a list of Face and/or Edge Objects from within the source Body used to create the Tessellation Object. First all specified Edges are rediscritized. Then any listed Face and the Faces touched by the retessellated Edges are retriangulated. Note that Quad Patches associated with Faces whose Edges were redone will be removed.
- parms** a set of 3 parameters that drive the Edge discretization and the Face triangulation. The first is the maximum length of an Edge segment or triangle side (in physical space). A zero is flag that allows for any length. The second is a curvature-based value that looks locally at the deviation between the centroid of the discrete object and the underlying geometry. Any deviation larger than the input value will cause the tessellation to be enhanced in those regions. The third is the maximum interior dihedral angle (in degrees) between triangle facets (or Edge segment tangents for a WIREBODY tessellation), note that a zero ignores this phase.
- icode** the integer return code

Redoes the discretization for specified Objects from within a Body Tessellation.

## Places the tessellation from one Body onto another

```
icode = EG_mapTessBody(ego tess, ego body, ego *newTess);  
icode = IG_mapTessBody(I*8 tess, I*8 body, I*8 newTess)  
newTess = tess.mapTessBody(body)  
newTess = mapTessBody(tess::Ego, body::Ego)
```

**tess** the Body Tessellation Object used to create the tessellation on **body**

**body** the Body Object (with a matching Topology) used to map the tessellation

**newTess** the returned resultant Tessellation of **body**. The triangulation is simply copied but the  $[u, v]$  and  $[x, y, z]$  positions reflect the input Body (**body**).

**icode** the integer return code

Maps the input discretization object to another Body Object. The topologies of the Body that created the input tessellation must match the topology of the body argument. The use of `EG_mapBody` (page 109) or `EG_mapBody2` (page 110) can assist. Can be useful for finite differences.

**Note:** Invoking `EG_moveEdgeVert` (page 126), `EG_deleteEdgeVert` (page 126) and/or `EG_insertEdgeVerts` (page 127) in the source tessellation before calling this routine invalidates the ability of `EG_mapTessBody` to perform this function.

## Provides a mapping for requested points

```

icode = EG_locateTessBody(const ego tess, int npts, const int *ifaces,
                        const double *uv, int *tris, double *weight);
icode = IG_locateTessBody(I*8 tess, I*4 npts, I*4 ifaces,
                        R*8 uv, I*4 tris, R*8 weight)

weight, tris = tess.locateTessBody(ifaces, uv, mapped=False)
weight, tris = locateTessBody(tess::Ego, ifaces::Int[],
                             uv::Float[] ([]), mapped::Bool)

```

- tess** the input Body Tessellation Object
- npts** the number of input requests
- ifaces** the face indices for each request; minus index refers to the use of a mapped Face index from EG\_mapBody (page 109) & EG\_mapTessBody (page 117) – **npts** in length
- uv** the  $[u, v]$  positions in the Face for each request –  $2*\mathbf{npts}$  in length
- tris** the resultant 1-bias triangle index – **npts** in length  
if input as **NULL** then this function will perform mapped evaluations
- weight** the vertex weights in the triangle that refer to the requested position (any negative weight indicates that the point was extrapolated) –or– the evaluated position based on the input **uvs** (when **tris** is **NULL**)  
 $3*\mathbf{npts}$  in length
- icode** the integer return code

Provides the triangle and the vertex weights for each of the input requests or the evaluated positions in a mapped tessellation.

## Gets the Edge discretization data

*ET, E1*

```

icode = EG_getTessEdge(const ego tess, int eIndex, int *len,
                      const double **xyzs, const double **ts);
icode = IG_getTessEdge(I*8 tess, I*4 eIndex, I*4 len,
                      R*8 xyzs, R*8 ts)

xyzs, ts = tess.getTessEdge(eIndex)
xyzs, ts = getTessEdge(tess::Ego, eIndex::Int)

```

- tess** the input Body Tessellation Object
- eIndex** the Edge index (1 bias). The Edge Objects and number of Edges can be retrieved via EG\_getBodyTopos (page 96) and/or EG\_indexBodyTopo (page 97). A minus index refers to the use of a mapped (+) Edge index from applying the functions EG\_mapBody (page 109) and EG\_mapTessBody (page 117).
- len** the returned number of vertices in the Edge discretization
- xyzs** the returned pointer to the set of coordinate data – 3\***len** in length
- ts** the returned pointer to the parameter values associated with each vertex – **len** in length
- icode** the integer return code

Note: DEGENERATE Edges return 2 vertices (both the same coordinates of the single Node) and the *t* range in **ts**. This Edge will not be referenced in the associated Face tessellation.

## Gets the Face triangulation data

ET, E1

```

icode = EG_getTessFace(const ego tess, int fIndex, int *len,
                      const double **xyz, const double **uv,
                      const int **ptype, const int **pindx, int *ntri,
                      const int **tris, const int **tric);
icode = IG_getTessFace(I*8 tess, I*4 fIndex, I*4 len,
                      R*8 xyz, R*8 uv,
                      I*4 ptype, I*4 pindx, I*4 *ntri,
                      I*4 tris, I*4 tric)
xyz, uv, ptype, pindx, tris, tric = tess.getTessFace(fIndex)
xyz, uv, ptype, pindx, tris, tric = getTessFace(tess::Ego, fIndex::Int)

```

- tess** the input Body Tessellation Object
- fIndex** the Face index (1 bias)
- len** the returned number of vertices in the Face triangulation
- xyz** the returned pointer to the set of coordinate data – 3\***len** in length
- uv** the returned pointer to the parameters for each vertex – 2\***len** in length
- ptype** returned pointer to the vertex type (-1 - internal, 0 - Node, > 0 Edge) – **len** in length
- pindx** returned pointer to vertex index (-1 internal) – **len** in length
- ntri** returned number of triangles
- tris** returned pointer to triangle indices, 3 per triangle (1 bias) – 3\***ntri** in length
- tric** returned pointer to neighbor information, 3 per triangle looking at opposing side: triangle (1-ntri), negative is Edge index for an external side – 3\***ntri** in length



## Gets the Discrete Loop data

ET, E1

```

icode = EG_getTessLoops(const ego tess, int fIndex, int *nloop,
                        const int **lIndices);
icode = IG_getTessLoops(I*8 tess, I*4 fIndex, I*4 nloop,
                        I*4 lIndices)

lIndices = tess.getTessLoops(fIndex)
lIndices = getTessLoops(tess::Ego, fIndex::Int)

```

**tess** the input Body Tessellation Object

**fIndex** the Face index (1 bias). The Face Objects and number of Faces can be retrieved via EG\_getBodyTopos and/or EG\_indexBodyTopo.

**nloop** the returned number of Loops in the Face triangulation

**lIndices** the returned pointer to a vector of the last index (bias 1) for each Loop – **nloop** in length. Notes:

- ① all boundary vertices are listed first for any Face tessellation,
- ② outer Loop data is ordered in the counter-clockwise direction, and
- ③ inner Loop(s) are ordered in the clockwise direction.

**icode** the integer return code

Retrieves the data for the Loops associated with the discretization of a Face from a Body-based Tessellation Object.

## Manually ask for quads

ET, E1

```
icode = EG_makeQuads(ego tess, double *qparms, int fIndex);  
icode = IG_makeQuads(I*8 tess, R*4 qparms, I*4 fIndex)  
    tess.makeQuads(qparms, fIndex)  
    makeQuads!(tess::Ego, qparms::Float[], fIndex::Int)
```

**tess** the input Body Tessellation Object

**qparms** a set of 3 parameters that drive the Quadrilateral patching for the Face. Any may be set to zero to indicate the use of the default value:

- **qparms[0]** – Edge matching tolerance expressed as the deviation from an aligned dot product [*default* : 0.05]
- **qparms[1]** – Maximum quad *side ratio* point count to allow [*default* : 3.0]
- **qparms[2]** – Number of smoothing loops [*default* : 0.0]

**fIndex** the Face index (1 bias)

**icode** the integer return code

Creates Quadrilateral Patches for the indicated Face and updates the Body-based Tessellation Object (if possible).

Note: you may want to consider using EG\_quadTess (page 115) instead.

## Ask for Faces with Quad Patches

ET, E1

```
icode = EG_getTessQuads(ego tess, int *nface, int **fList);  
icode = IG_getTessQuads(I*8 tess, I*4 nface, I*4 fList)  
fList = tess.getTessQuads()  
fList = getTessQuads(tess::Ego)
```

- tess** the Body Tessellation Object
- nface** the returned number of Faces with Quad patches
- fList** the returned pointer the Face indices (1 bias) – **nface** in length (freeable)  
The Face Objects themselves can be retrieved via EG\_getBodyTopos
- icode** the integer return code

Returns a list of Face indices found in the Body-based Tessellation Object that has been successfully Quadded via EG\_makeQuads – page 122.

## Gets the Quad Face data

ET, E1

```

icode = EG_getQuads(const ego tess, int fIndex, int *len,
                    const double **xyz, const double **uv,
                    const int **ptype, const int **pindx, int *npatch);
icode = IG_getQuads(I*8 tess, I*4 fIndex, I*4 len,
                    R*8 xyz, R*8 uv,
                    I*4 ptype, I*4 pindx, I*4 *npatch)
xyz, uv, ptype, pindx, npatch = tess.getQuads(fIndex)
xyz, uv, ptype, pindx, npatch = getQuads(tess::Ego, fIndex::Int)

```

**tess** the input Body Tessellation Object  
**fIndex** the Face index (1 bias)  
**len** the returned number of vertices in the Face triangulation  
**xyz** the returned pointer to the set of coordinate data – 3\***len** in length  
**uv** the returned pointer to the parameters for each vertex – 2\***len** in length  
**ptype** returned pointer to the vertex type (-1 - internal, 0 - Node, > 0 Edge) – **len** in length  
**pindx** returned pointer to vertex index (-1 internal) – **len** in length  
**npatch** returned number of patches  
**icode** the integer return code

Retrieves the data associated with the Quad-patching of a Face (using EG\_makeQuads – page 122) in a Body-based Tessellation Object.

## Gets the Quad Face patch data

ET, E1

```

icode = EG_getPatch(const ego tess, int fIndex, int pIndex,
                    int *n1, int *n2, const int **pvindex,
                    const int **pbounds);
icode = IG_getPatch(I*8 tess, I*4 fIndex, I*4 pIndex,
                    I*4 n1, I*4 n2, CPTR pvindex,
                    CPTR pbounds)
n1, n2, pvindex, pbounds = tess.getPatch(fIndex, pIndex)
n1, n2, pvindex, pbounds = getPatch(tess::Ego, fIndex::Int,
                                     pIndex::Int)

```

- tess** the input Body Tessellation Object
- fIndex** the Face index (1 bias)
- pIndex** the Patch index (from 1 to **npatch** returned by EG\_getQuads)
- n1** the returned patch size in the first direction (indexed by *i*)
- n2** the returned patch size in the second direction (indexed by *j*)
- pvindex** the returned pointer to  $n1 * n2$  indices that define the patch
- pbounds** returned pointer to the neighbor bounding information for the patch. The first represents the segments at the base (*j* at base and increasing in *i*), the next is at the right (with *i* at max and *j* increasing). The third is the top (with *j* at max and *i* decreasing) and finally the left (*i* at min and *j* decreasing).  $(2 * (n1 - 1) + 2 * (n2 - 1))$  in length

Retrieves the patch data associated with the Quad data on a Face (generated by EG\_makeQuads – page 122) in a Body-based Tessellation Object.

## Move the position of an Edge Vertex

*ET, E1*

```
icode = EG_moveEdgeVert(ego tess, int eIndex, int vIndex, double t);
icode = IG_moveEdgeVert(I*8 tess, I*4 eIndex, I*4 vIndex, R*8 t)
    tess.moveEdgeVert(eIndex, vIndex, t)
    moveEdgeVert!(tess::Ego, eIndex::Int, vIndex::Int, t::Float)
```

**tess** the Body Tessellation Object (not on WIREBODIES)  
**eIndex** the Edge index (1 bias)  
**vIndex** the Vertex index in the Edge (2 to **nVert**-1)  
**t** the new parameter value on the Edge for the point

Note: Will invalidate the Quad patches on any Faces touching the Edge.

## Deletes an Edge Vertex

*ET, E1*

```
icode = EG_deleteEdgeVert(ego tess, int eIndex, int vIndex, int dir);
icode = IG_deleteEdgeVert(I*8 tess, I*4 eIndex, I*4 vIndex, I*4 dir)
    tess.deleteEdgeVert(eIndex, vIndex, dir)
    deleteEdgeVert!(tess::Ego, eIndex::Int, vIndex::Int, dir::Int)
```

**tess** the Body Tessellation Object (not on WIREBODIES)  
**eIndex** the Edge index (1 bias)  
**vIndex** the Vertex index in the Edge (2 to **nVert**-1)  
**dir** the direction to collapse any triangles (either -1 or 1)

Note: Will invalidate the Quad patches on any Faces touching the Edge.

## Insert vertices on an Edge

*ET, E1*

```

icode = EG_insertEdgeVerts(ego tess, int eIndex, int vIndex, int len,
                           double *ts);
icode = IG_insertEdgeVerts(I*8 tess, I*4 eIndex, I*4 vIndex, I*4 len,
                           R*8      ts)
tess.insertEdgeVerts(eIndex, vIndex, ts)
insertEdgeVerts!(tess::Edge, eIndex::Int, vIndex::Int, ts::Float([]))

```

- tess** the Body Tessellation Object (not on WIREBODIES)
- eIndex** the Edge index (1 bias)
- vIndex** the Vertex index in the Edge (2 to **nVert**-1)
- len** the number of points to insert
- ts** a vector of *t* values for the new points. Must be monotonically increasing and be greater than the *t* of **vIndex** and less than the *t* of **vIndex**+1.
- icode** the integer return code

Note: Will invalidate the Quad patches on any Faces touching the Edge.

## Open an existing Tessellation Object

*ET, E1*

```
icode = EG_openTessBody(ego tess);  
icode = IG_openTessBody(I*8 tess)  
    tess.openTessBody()  
    openTessBody(tess::Ego)
```

**tess** the Tessellation Object to open

**icode** the integer return code

Opens an existing Tessellation Object for replacing Edge/Face discretizations.

## Open a new (empty) Tessellation Object

*ET, E1*

```
icode = EG_initTessBody(ego body, ego *tess);  
icode = IG_initTessBody(I*8 body, I*8 tess)  
    tess = body.initTessBody()  
    tess = initTessBody(body::Ego)
```

**body** the input object, may be any Body type

**tess** resultant empty Tessellation Object where each Edge in the BODY must be filled via a call to EG\_setTessEdge (page 130) and each Face must be filled with invocations of EG\_setTessFace (page 131). The Tessellation Object is considered open until all Edges have been set (for a WIREBODY), all Faces have been set (for other Body types) or EG\_finishTess (page 133) is called.

**icode** the integer return code



## Status of a Tessellation Object

*ET, E1*

```

icode = EG_statusTessBody(ego tess, ego *body, int *stat, int *npts);
icode = IG_statusTessBody(I*8 tess, I*8 body, I*4 stat, I*4 npts)
body, stat, icode, npts = tess.statusTessBody()
body, stat, icode, npts = statusTessBody(tess::Ego)

```

- tess** the Tessellation Object to query
- body** the returned associated Body Object
- stat** the returned state of the tessellation: -1 – closed but warned, 0 – open, 1 – OK, 2 – displaced
- npts** the returned number of global points in the tessellation (0 – open)
- icode** the integer return code: EGADS\_SUCCESS – complete, EGADS\_OUTSIDE – still open

Note: Placing the attribute “.mixed” on **tess** before invoking this function allows for tri/quad (2 tris) tessellations. The type must be ATTRINT and the length is the number of Faces, where the values are the number of quads (triangle pairs) per Face. Single triangles are followed by triangle pairs for a Face with both triangle and quads.

Given quad 1 2 3 4 ==>	4---3
trias 1 2 3 and 1 3 4	/
	1---2

## Sets the Edge discretization data

ET, E1

```

icode = EG_setTessEdge(ego tess, int eIndex, int len,
                      const double *xyzs, const double *ts);
icode = IG_setTessEdge(I*8 tess, I*4 eIndex, I*4 len,
                      R*8 xyzs, R*8 ts)
tess.setTessEdge(eIndex, xyzs, ts)
setTessEdge!(tess::Ego, eIndex::Int, xyzs::Float[][], ts::Float[])

```

- tess** the open Tessellation Object
- eIndex** the Edge index (1 bias). The Edge Objects and number of Edges can be retrieved via EG\_getBodyTopos and/or EG\_indexBodyTopo. If this Edge already has assigned data, it is overwritten.
- len** the number of vertices in the Edge discretization
- xyzs** the pointer to the set of coordinate data – 3\*len in length
- ts** the pointer to the parameter values associated with each vertex – len in length
- icode** the integer return code

## Notes:

- 1 all vertices must be specified in increasing  $t$
- 2 the coordinates for the first and last vertex MUST match the appropriate Node's coordinates
- 3 problems are reported to *Standard Out* regardless of the OutLevel

## Sets the Face triangulation data 1/2

ET, E1

```

icode = EG_setTessFace(ego tess, int fIndex, int len,
                      const double *xyz, const double *uv,
                      int ntri, const int *tris);
icode = IG_setTessFace(I*8 tess, I*4 fIndex, I*4 len,
                      R*8 xyz, R*8 uv,
                      I*4 ntri, I*4 tris)
tess.setTessFace(fIndex, xyz, uv, tris)
setTessFace!(tess::Ego, fIndex::Int, xyz::Float[][],
             uv::Float[][], tris::Int[][])

```

- tess** the open Body Tessellation Object
- fIndex** the Face index (1 bias). The Face Objects and number of Faces can be retrieved via EG\_getBodyTopos and/or EG\_indexBodyTopo. If this Face already has assigned data, it is overwritten.
- len** the number of vertices in the Face triangulation
- xyz** the pointer to the set of coordinate data – 3\*len in length
- uv** the pointer to the parameters for each vertex – 2\*len in length
- ntri** returned number of triangles
- tris** the pointer to triangle indices, 3 per triangle (1 bias) – 3\*ntri in length
- icode** the integer return code

## Sets the Face triangulation data 2/2

*ET, E1*

### Notes:

- 1 All Edges associated with the Face must have been set
- 2 Any vertex associated with a Node or an Edge must use the exact coordinates specified by the Node or the Edge discretization
- 3 During the execution of `EG_setTessFace` the vertices that are input will go through renumbering. This is because there is an internal assumption that the first suite of coordinates represent the Edge tessellation ordered by the Loops found in the Face. This is usually fine, but there are circumstances that you may need to know the mapping, so that later you can find specific a specific vertex. This can be reverse engineered by maintaining the initial triangle list, calling `EG_getTessFace` – page 120 (after the invocation of `EG_setTessFace`) and using the triangles to construct the index mapping. To state it in another way: the triangle order is fixed but the list of vertices has been rearranged.
- 4 Problems are reported to *Standard Out* regardless of the `OutLevel`

## Finish up and close an Open Tessellation Object *ET, E1*

```
icode = EG_finishTess(ego tess, double *parms);  
icode = IG_finishTess(I*8 tess, R*8      parms)  
    tess.finishTess(parms)  
    finishTess!(tess::Ego, parms::Float[])
```

**tess** the Open Tessellation Object to close

**parms** a set of 3 parameters that drive the Edge discretization and the Face triangulation. The first is the maximum length of an Edge segment or triangle side (in physical space). A zero is flag that allows for any length. The second is a curvature-based value that looks locally at the deviation between the centroid of the discrete object and the underlying geometry. Any deviation larger than the input value will cause the tessellation to be enhanced in those regions. The third is the maximum interior dihedral angle (in degrees) between triangle facets (or Edge segment tangents for a WIREBODY tessellation), note that a zero ignores this phase.

**icode** the integer return code

Completes the discretization for unfilled entities found within the input Tessellation Object.

Note: an open Tessellation Object is created by `EG_initTessBody` (page 128) and can be partially filled via `EG_setTessEdge` (page 130) and/or `EG_setTessFace` (page 131) before this function is invoked.

## Global Lookup

ET, E1

```

icode = EG_localToGlobal(const ego tess, int ind, int locl, int *gbl);
icode = IG_localToGlobal(I*8 tess, I*4 ind, I*4 locl, int gbl)
    gbl = tess.localToGlobal(ind, locl)
    gbl = localToGlobal(tess::Ego, ind::Int, locl::Int)
    tess the closed Tessellation Object
    ind the topological index (1 bias) – 0 Node, (-) Edge, (+) Face
    locl the local (or Node) index
    gbl the returned global vertex index

```

## Gets the vertex type and index

ET, E1

```

icode = EG_getGlobal(const ego tess, int global, int *pytpe,
                    int *pindex, double *xyz);
icode = IG_getGlobal(I*8 tess, I*4 global, I*4 pytpe,
                    I*4 pindex, R*8 xyz)
pytpe, pindex, xyz = tess.getGlobal(global)
pytpe, pindex, xyz = getGlobal(tess::Ego, global::Int)
    tess the closed Tessellation Object
    global the global index (1 bias)
    pytpe the point type (-) Face local index, (0) Node, (+) Edge local index
    pindex the point topological index (1 bias)
    xyz the filled (3 in length) coordinates at this global index (can be NULL)

```

## Returns the Discrete Mass Properties

dot

```
icode = EG_tessMassProps(const ego tess, double *props);  
icode = IG_tessMassProps(I*8 tess, R*8 props)  
volume, aeraOrLen, CG, I = tess.tessMassProps()  
volume, aeraOrLen, CG, I = tessMassProps(tess::Ego)
```

- tess** the Body Tessellation Object used to compute the Mass Properties
- props** 14 **doubles** filled reflecting Volume, Area (or Length), Center of Gravity (3) and the inertia matrix at CG (9)
- icode** the integer return code

Computes and returns the physical and inertial properties of a Tessellation Object.

## Write Tessellation to Disk – Deprecated

```
icode = EG_saveTess(ego tess, const char *filename);  
icode = IG_saveTess(I*8 tess, C**      filename)
```

**tess** the closed Tessellation Object

**filename** the filename (including extension)

**icode** the integer return code

The extension can be anything, but it is suggested that “.eto” (EGADS Tessellation Object) be used.

## Read Tessellation from Disk – Deprecated

```
icode = EG_loadTess(ego body, const char *filename, ego *tess);  
icode = IG_loadTess(I*8 body, C**      filename, I*8 tess)
```

**body** the Body Object to attach the tessellation from the file

**filename** the filename (including extension) to load

**tess** the returned Tessellation Object

**icode** the integer return code



## Perform the Solid Boolean Operations – Deprecated

```
icode = EG_solidBoolean(const ego src, const ego tool, int oper,  
                        ego *model);  
icode = IG_solidBoolean(I*8 src, I*8 tool, I*4 oper,  
                        I*8 model)
```

**src** the source SOLIDBODY Body Object

**tool** the tool object:

either a SOLIDBODY for all operators –or–  
a FACE/FACEBODY for SUBTRACTION

**oper** one of: INTERSECTION, SUBTRACTION, FUSION

**model** the resultant Model Object (this is because there may be multiple bodies from either the SUBTRACTION or INTERSECTION operation).

**icode** the integer return code

Performs the Solid Boolean Operations (SBOs) on the source Body Object (that has the type SOLIDBODY). The tool Object types depend on the operation. This supports Intersection, Subtraction and Union.

Note: This may be called with **src** being a Model Object. In this case **tool** may be a SOLIDBODY for Intersection/Subtraction or a FACE/FACEBODY for Fusion. The input Model Object may contain anything, but must not have duplicate topology.

## Perform the Boolean Operations

```
icode = EG_generalBoolean(ego src, ego tool, int oper, double tol,  
                           ego *model);  
icode = IG_generalBoolean(I*8 src, I*8 tool, I*4 oper, R*8 tol,  
                           I*8 model)  
model = src.generalBoolean(tool, oper, tol=0.0)  
model = generalBoolean(src::Ego, tool::Ego, oper::Int; tol::Real)
```

**src** the source in the form of a Model or Body Object

**tool** the tool in the form of a Model or Body Object

**oper** one of: INTERSECTION, SUBTRACTION, FUSION, SPLITTER

**tol** the tolerance applied when performing the operation. If set to 0.0, the minimum tolerance is applied

**model** the returned resultant Model Object (this is because there may be multiple bodies from either the SUBTRACTION or INTERSECTION operation).

**icode** the integer return code

Performs the Boolean Operations on the source Body Object(s). The tool Body Object(s) are applied depending on the operation. This supports Intersection, Splitter, Subtraction and Union.

Note: The Body Object(s) for **src** and **tool** can be of any type and the results depend on the operation.

## Perform a Union on Sheet Bodies

```
icode = EG_fuseSheets(ego src, ego tool, ego *sheet);  
icode = IG_fuseSheets(I*8 src, I*8 tool, I*8 sheet)  
sheet = src.generalBoolean(tool)  
sheet = fuseSheets(src::Ego, tool::Ego)
```

**src** the source Sheet Body Object

**tool** the tool Sheet Body Object

**sheet** the returned resultant Body Object (mttype is SHEETBODY).

**icode** the integer return code

Fuses (unions) two Sheet Body Objects resulting in a single Sheet Body Object.

## Provide the Intersection

```
icode = EG_intersection(ego src, ego tool, int *nEdge, ego **pairs,  
                        ego *model);  
icode = IG_intersection(I*8 src, I*8 tool, I*4 nEdge, I*8 pairs,  
                        I*8 model)  
pairs, model = src.intersection(tool)  
pairs, model = intersection(src::Ego, tool::Ego)
```

**src** the source Body Object

**tool** the FACE/FACEBODY/SHEETBODY/SOLIDBODY tool Object

**nEdge** the returned number of Edge Objects created

**pairs** the returned pointer to Face/Edge Object pairs – 2\*nEdge in len (freeable)  
can be **NULL** (if you don't need the data – the Edge Objects are in **model**)

**model** the resultant Model Object which contains the set of WIREBODY Objects (this is  
because there may be multiple Loop Objects as a result of the operation). Deleting  
**model** invalidates the data in **pairs**.

**icode** the integer return code

Intersects the source Body Object (that has the type SOLIDBODY, SHEETBODY or FACEBODY) with surface or surfaces. The tool Object contains the intersecting geometry in the form of a FACEBODY, SHEETBODY, SOLIDBODY or a single Face.

Note: The Edge Objects contained within the Loops have the attributes of the Face Object in **src** responsible for that Edge.

## Scribe a Body

```
icode = EG_imprintBody(ego src, int nObj, ego *pairs, ego *result);  
icode = IG_imprintBody(I*8 src, I*4 nObj, I*8 pairs, I*8 result)  
result = src.imprintBody(pairs)  
result = imprintBody(src::Ego, pairs::Ego[[]])
```

- src** the source Body Object
- nObj** the number of Object pairs to imprint
- pairs** pointer to a list of Face/Edge and/or Face/Loop Object pairs to scribe  
2\*nObj in len – can be the output from EG\_intersection.
- result** the returned resultant Body Object (with the same type as **src**),  
though the splitting of Face Body Object results in a SHEETBODY
- icode** the integer return code

Imprints Edge/Loop Objects on the source Body Object (that has the type SOLIDBODY, SHEETBODY or FACEBODY). The Edge/Loop Objects are paired with the Faces in the source that will be scribed.

Note: Under rare circumstance this may fail and return the indication of success [especially after seeing the message: *EGADS Info: splitBody = xx – using OpenCASCADE (EG\_imprintBody)!.*

If appropriate, you may want to consider using the SPLITTER operation of EG\_generalBoolean.

## Fillet a Body

```
icode = EG_filletBody(ego src, int nEdge, ego *edges, double radius,
                      ego *result, int **maps);
icode = IG_filletBody(I*8 src, I*4 nEdge, I*8 edges, R*8 radius,
                      I*8 result, CPTR maps)
result, map = src.filletBody(edges, radius)
result, map = filletBody(src::Ego, edges::Ego[], radius::Float)
```

- src** the source Body Object
- nEdge** the number of Edge Objects to fillet
- edges** pointer to a list of Edges to fillet – **nEdge** in len
- radius** the radius of the fillets created
- result** the resultant Body Object (with the same type as **src**)
- maps** the returned pointer to a list of Face mappings (in **result**) which includes operations (see page 13) and an index in **src** where the Face originated –  $2 * (\text{number of Faces in result})$  in length (freeable)
- icode** the integer return code

Fillets the Edges on the source Body Object (that has the type SOLIDBODY or SHEETBODY).

## Chamfer a Body

```

icode = EG_chamferBody(ego src, int nEdge, ego *edges, ego *faces,
                      double dis1, double dis2, ego *result,
                      int **maps);

icode = IG_chamferBody(I*8 src, I*4 nEdge, I*8 edges, I*8 faces,
                      R*8 dis1, R*8 dis2, I*8 result,
                      CPTR maps)

result, map = src.chamferBody(edges, faces, dis1, dis2)
result, map = chamferBody(src::Ego, edges::Ego[], faces::Ego[],
                          dis1::Float, dis2::Float)

```

- src** the source Body Object
- nEdge** the number of Edge Objects to chamfer
- edges** pointer to a list of Edges to chamfer – **nEdge** in len
- faces** pointer to a list of Face Objects to to measure **dis1** from – **nEdge** in len
- dis1** the distance from the Face Object to chamfer
- dis2** the distance from the *other* Face Object to chamfer
- result** the resultant Body Object (with the same type as **src**)
- maps** the returned pointer to a list of Face mappings (in **result**) which includes operations (see page 13) and an index in **src** where the Face originated –  $2 * (\text{number of Faces in } \text{result})$  in length (freeable)
- icode** the integer return code

Chamfers the Edges on the source Body Object (that has the type SOLIDBODY or SHEETBODY).

## Hollow a Body

```
icode = EG_hollowBody(ego src, int nFace, ego *faces, double off,
                      int join, ego *result, int **maps);
icode = IG_hollowBody(I*8 src, I*4 nFace, I*8 faces, R*8 off,
                      I*4 join, I*8 result, CPTR maps)
result, map = src.hollowBody(faces, off, join)
result, map = hollowBody(src::Ego, faces::Ego([]), off::Float,
                          join::Int)
```

- src** the source Body Object (SOLIDBODY, SHEETBODY, FACEBODY or FACE\*)
- nFace** the number of Face Objects to remove (0 performs an *offset*)
- faces** pointer to a list of FACE objects to remove – **nFace** in len
- off** the wall thickness (or offset) of the result
- join** 0 – fillet-like corners, 1 – expanded corners
- result** the resultant Body Object
- maps** the returned pointer to a list of Face mappings (in **result**) which includes operations (see page 13) and an index in **src** where the Face originated –  $2 * (\text{number of Faces in result})$  in length (freeable)
- icode** the integer return code

A hollowed *solid* is built from an initial SOLIDBODY Object and a set of Faces that bound the *solid*. These Faces are removed and the remaining become the walls of the *hollowed solid* with the specified thickness. If there are no Faces specified then the Body is *offset* by **off** (which can be negative).

\* Note: If **src** is a Face, then **faces** should be a list of Edges and the result will be a Face Object. **maps** in this case is not filled.



## Revolve to create a Body

```
icode = EG_rotate(ego src, double angle, double *axis, ego *result);  
icode = IG_rotate(I*8 src, R*8 angle, R*8 axis, I*8 result)  
result = src.rotate(angle, axis)  
result = rotate(src::Ego, angle::Float, axis::Float[]([]))
```

- src** the source Body Object (not SHEETBODY or SOLIDBODY)
- angle** the angle to rotate the object through [0-360 Degrees]
- axis** pointer to a point (on the axis) and a direction (6 in length)
- result** the returned resultant Body Object (type is one higher than **src**)
- icode** the integer return code

Revolves the source Object about the axis through the angle specified. If the Object is either a Loop or WIREBODY the result is a SHEETBODY. If the source is either a Face or FACEBODY then the returned Body Object is a SOLIDBODY.

## Extrude to create a Body

dot

```
icode = EG_extrude(ego src, double dist, double *dir, ego *result);  
icode = IG_extrude(I*8 src, R*8 dist, R*8 dir, I*8 result)  
result = src.extrude(dist, dir)  
result = extrude(src::Ego, dist::Float, dir::Float[])
```

**src** the source Body Object (not SHEETBODY or SOLIDBODY)

**dist** the distance to extrude

**dir** pointer to the vector that is the extrude direction (3 in length)

**result** the returned resultant Body Object (type is one higher than **src**)

**icode** the integer return code

Extrudes the source Object through the distance and direction specified. If the Object is either a Loop or WIREBODY the result is a SHEETBODY. If the source is either a Face or FACEBODY then the returned Body Object is a SOLIDBODY.

## Sweep to create a Body

```
icode = EG_sweep(ego src, ego spine, int mode, ego *result);  
icode = IG_sweep(I*8 src, I*8 spine, I*4 mode, I*8 result)  
result = src.sweep(spine, mode)  
result = sweep(src::Ego, spine::Ego, mode::Int)
```

**src** the source Body Object (not SHEETBODY or SOLIDBODY)

**spine** the Object used as guide curve segment(s) to sweep the source through

**mode** sweep mode:

0	Corrected Frenet	5	Guide AC
1	Fixed	6	Guide Plan
2	Frenet	7	Guide AC With Contact
3	Constant Normal	8	Guide Plan With Contact
4	Darboux	9	Discrete Trihedron

**result** the returned resultant Body Object (type is one higher than **src**)

**icode** the integer return code

Sweeps the source Object through the “spine” specified. The spine can be either an Edge, Loop or WIREBODY Object. If the source Object is either a Loop or WIREBODY the result is a SHEETBODY. If the source is either a Face or FACEBODY then the returned Object is a SOLIDBODY.

**Note:** this does not always work as expected...

## Lofts through sections to create a Body – Deprecated

```
icode = EG_loft(int nSection, ego *sections, int option, ego *result);  
icode = IG_loft(I*4 nSection, I*8 sections, I*4 option, I*8 result)
```

- nSection** the number of Sections in the Loft Operation
- sections** pointer to a list of WIREBODY or Loop Objects to Loft – **nSection** in len  
the first and last can be Node Objects
- option** bit flag that controls the loft:  
1 – SOLIDBODY result (default is SHEETBODY)  
2 – Ruled (linear) Loft (default is smooth)
- result** the resultant Body Object
- icode** the integer return code

Lofts the input Objects to create a Body Object (that has the type SOLIDBODY or SHEETBODY).

Please use either `EG_blend` (page 150) or `EG_ruled` (page 149).

## Linearly lofts through sections to create a Body

dot

```
icode = EG_ruled(int nSection, ego *sections, ego *result);  
icode = IG_ruled(I*4 nSection, I*8 sections, I*8 result)  
result = egads.ruled(sections)  
result = ruled(sections::Ego)
```

**nSection** the number of Sections in the *rule* Operation

note: interior repeated sections are ignored

**sections** a list of FACEBODY, Face, WIREBODY or Loop Objects (**nSection** in len)  
FACEBODY/Faces must have only a single Loop; all or the first and last sections can be Nodes; if the first/last are Nodes and/or FACEBODY/Faces (and the internal sections are CLOSED) the result will be a SOLIDBODY otherwise it will be a SHEETBODY; if the first and last sections contain equivalent Loops (and all sections are CLOSED) a periodic (torus-like) SOLIDBODY is created

**result** the resultant Body Object

**icode** the integer return code

Produces a Body Object (that has the type SOLIDBODY or SHEETBODY) that goes through the sections by *ruling* surfaces between each. All sections must have the same number of Edges (except for Nodes) and the Edge order in each is used to specify the connectivity. An exception is when the attribute “multiNode” is found on Nodes in the Loop, which needs to be numeric and indicates the multiplicity of the Node [creating Degenerate Edge(s) in the result]. If this is the case, the number of Edges and the sum of the Node multiplicities must be the same for each section. There is a special case when the attributed Node is the first in the Loop and the Loop is closed: a second numeric value is required in the attribute where this one indicates the the start – 0 is all degenerates are at the start, and if the number of multiples then the start is the Edge itself, where the degenerates end up at the end of the Loop.

Note: for both EG\_blend and EG\_ruled: all Loops must have their Edges ordered in a CCW (counterclockwise) manner.

## Lofts through sections to create a Body

dot

```
icode = EG_blend(int nSection, ego *sections, double *rc1,
                double *rc2, ego *result);
icode = IG_blend(I*4 nSection, I*8 sections, R*8 rc1,
                R*8 rc2, I*8 result)
result = egads.blend(sections, rc1=None, rc2=None)
result = blend(sections::Ego[]; rc1::Float[], rc2::Float[])
```

**nSection** the number of Sections in the *blend* Operation

**sections** a list of FACEBODY, Face, WIREBODY or Loop Objects (**nSection** in len)  
 FACEBODY/Faces must have only a single Loop; all or the first and last sections can be Nodes; if the first/last are Nodes and/or FACEBODY/Faces (and the internal sections are CLOSED) the result will be a SOLIDBODY otherwise it will be a SHEETBODY; if the first and last sections contain equivalent Loops (and all sections are CLOSED) a periodic ( $C^0$  at closure) SOLIDBODY is created.  
 Interior sections are  $C^2$  and can be repeated once<sup>†</sup> for  $C^1$  or twice for  $C^0$

**rc1** specifies treatment<sup>‡</sup> at the first section (or **NULL** for no treatment)

**rc2** specifies treatment<sup>‡</sup> at the last section (or **NULL** for no treatment)

**result** the resultant Body Object

*Lofts* the input Objects to create a Body Object (that has the type SOLIDBODY or SHEETBODY). Cubic BSplines are used. All sections must have the same number of Edges (except for single Nodes) and the Edge order in each (in a CCW manner) is used to set the connectivity.

<sup>†</sup> section attribute .C1side set to “Fwd” or “rev” overrides the default side to compute the  $C^1$  tangency.

<sup>‡</sup> for Nodes – elliptical treatment (8 in length): radius of curvature1, unit dir, radius of curvature2, orthogonal dir (**nSection** must be at least 3 or 4 for treatments at both ends); for other sections – setting tangency (4 in length): magnitude, unit direction; for Faces with 2 or 3 Edges in the Loop – make a *Wing Tip-like* cap: zero, growthFactor (length of 2).

Initialize *Effective Topology* Body

ET

```
icode = EG_initEBody(ego tess, double angle, ego *ebody);  
icode = IG_initEBody(I*8 tess, R*8 angle, I*8 ebody)  
ebody = tess.initEBody(angle)  
ebody = initEBody(tess::Ego, angle::Float)
```

- tess** the input Solid or Sheet Body Tessellation Object (can be quite coarse)
- angle** angle used to determine if Nodes on open Edges of Sheet Bodies can be removed. The dot of the tangents at the Node is compared to this angle (in degrees). If the dot is greater than the angle, the Node does not disappear. The angle is also used to test Edges to see if they can be removed. Edges with normals on both trimmed Faces showing deviations greater than the input angle will not disappear. Valid range 0.0 to 90.0 and should be closer to zero.
- ebody** the resultant Open *Effective Topology* Body Object
- icode** the integer return code

Takes as input a Body Tessellation Object and returns an Open EBody fully initialized with *Effective Objects* (that may only contain a single *non-effective* object). EEdges are automatically merged where possible (removing Nodes that touch 2 Edges, unless degenerate or marked as “.Keep”). The tessellation is used (and required) in order that single UV space be constructed for EFaces.

## Construct EFaces using an Attribute

ET

```

icode = EG_makeAttrEFaces(ego ebody, const char *attrName,
                           int *nface, ego **efaces);
icode = IG_makeAttrEFaces(I*8 ebody, C** attrName,
                           I*4 nface, I*8 efaces)
efaces = ebody.makeAttrEFaces(attrName)
efaces = makeAttrEFaces(ebody::Ego, attrName::String)

```

- ebody** the input Open *Effective Topology* Body Object
- attrName** the attribute name used to collect Faces into an EFaces. The attribute value(s) are then matched to make the collections.
- nface** the returned number of constructed EFaces
- efaces** the returned pointer to a list of EFaces constructed (freeable) – may be **NULL**
- icode** the integer return code

Modifies the EBody by finding “free” Faces (a single Face in an EFace) with **attrName** and the same attribute value(s), thus making a collection of EFaces. All attributes matching in the collection of Faces are moved to the EFace(s) (at a minimum **attrName** will persist) unless in “Full Attribute” mode, which then performs attribute merging. This function returns the number of EFaces possibly constructed (**nface**). The *UVbox* can be retrieved via calls to either `EG_getTopology` (page 90) or `EG_getRange` (page 81) with the returned appropriate **efaces**.

Note: triangulations must *touch* to be within an EFace.



## Construct an EFace

ET

```

icode = EG_makeEFace(ego ebody, int nface, ego *faces, ego *eface);
icode = IG_makeEFace(I*8 ebody, I*4 nface, I*8 faces, I*8 eface)
eface = ebody.makeEFace(faces)
eface = makeEFace(ebody::Ego, faces::Ego([]))

```

**ebody** the input Open *Effective Topology* Body Object  
**nface** the number of Face Objects in **faces**  
**faces** the list of Face Objects used to make **eface** (**nface** in length)  
**eface** the returned constructed EFace Object now in **ebody**  
**icode** the integer return code

Modifies the EBody by removing the **nface** “free” Faces and constructing a single **eface** (returned for convenience – the **ebody** is updated). All attributes matching in **faces** are moved to **eface** unless the “Full Attribution” mode is in place. This constructs a single UV for the **faces**. The *UVbox* can be retrieved via calls to `EG_getTopology` (page 90) or `EG_getRange` (page 81) with **eface**.

## Finish an EBody

ET

```

icode = EG_finishEBody(ego ebody);
icode = IG_finishEBody(I*8 ebody)
ebody.finishEBody()
finishEBody!(ebody::Ego)

```

**ebody** the input Open *Effective Topology* Body Object that will be Closed

## Parameter/Object Lookup

ET, E1

```
icode = EG_effectiveMap(ego eobj, double *eparam, ego *obj,  
                        double *param);  
icode = IG_effectiveMap(I*8 eobj, R*8      eparam, I*8  obj,  
                        R*8      param)  
obj, param = eobj.effectiveMap(eparam)  
obj, param = effectiveMap(eobj::Ego, eparam::Float([]))
```

- eobj** the input *Effective Topology* Object (either EEdge or EFace)
- eparam**  $t$  for EEdges / the  $[u, v]$  for an EFace
- obj** the returned source Object in the Body
- param** the returned  $t$  for an Edge / the returned  $[u, v]$  for the Face
- icode** the integer return code

Returns the evaluated location in the BRep for the *Effective Topology* entity.

## Returns details of an EEdge

*ET*, E1

```
icode = EG_effectiveEdgeList(ego edge, int *nedge, ego **edges,
                             int **senses, double **tstart);
icode = IG_effectiveEdgeList(I*8 edge, I*4 nedge, I*8 edges,
                             I*4 senses, R*8 tstart)

edges, senses, tstart = eedge.effectiveEdgeList()
edges, senses, tstart = effectiveEdgeList(eedge::Ego)
```

- eedge** the *Effective Topology* Edge Object
- nedge** the returned number of Edges in **eedge** and others
- edges** the returned pointer to a list of Edges – **nedge** in length (freeable)
- senses** the returned pointer to a list of senses – **nedge** in length (freeable)
- tstart** the returned pointer to a list of  $t$  starting values – **nedge** in length (freeable)
- icode** the integer return code

Returns the list of Edge entities in the source Body that make up the EEdge. A pointer to an integer list of senses for each Edge is returned as well as the starting  $t$  value in the EEdge (remember that the  $t$  will go in the opposite direction in the Edge if the sense is SREVERSE).

EG_addKnots	78	EG_fitTriangles	77
EG_alloc	63	EG_flipObject	57
EG_approximate	76	EG_free	63
EG_arcLength	81	EG_fuseSheets	139
EG_attributeAddSeq	70	EG_generalBoolean	138
EG_attributeAdd	64	EG_getArea	98
EG_attributeDel	65	EG_getBodyTopos	96
EG_attributeDup	68	EG_getBody	105
EG_attributeGet	67	EG_getBoundingBox	98
EG_attributeNumSeq	71	EG_getContext	61
EG_attributeNum	65	EG_getEdgeUV	101
EG_attributeRetSeq	72	EG_getGeometry	74
EG_attributeRet	66	EG_getGlobal	134
EG_blend	150	EG_getInfo	60
EG_calloc	63	EG_getMassProperties	99
EG_chamferBody	143	EG_getPatch	125
EG_close	52	EG_getQuads	124
EG_contextCopy	59	EG_getRange	81
EG_convertToBSpline	86	EG_getTessEdge	119
EG_convertToBSplineRange	87	EG_getTessFace	120
EG_copyObject	58	EG_getTessGeom	113
EG_curvature	84	EG_getTessLoops	121
EG_deleteEdgeVert	126	EG_getTessQuads	123
EG_deleteObject	52	EG_getTolerance	105
EG_effectiveEdgeList	155	EG_getTopology	90
EG_effectiveMap	154	EG_getTransform	57
EG_evaluate	82	EG_getWindingAngle	111
EG_exportModel	62	EG_hollowBody	144
EG_extrude	146	EG_importModel	62
EG_filletBody	142	EG_imprintBody	141
EG_finishEBody	153	EG_inFace	100
EG_finishTess	133	EG_inTopology	100

EG_indexBodyTopo	97	EG_matchBodyEdges	107
EG_initEBody	151	EG_matchBodyFaces	108
EG_initTessBody	128	EG_mergeBSplineCurves	86
EG_insertEdgeVert	127	EG_moveEdgeVert	126
EG_intersection	140	EG_objectBodyTopo	97
EG_invEvaluate	83	EG_openTessBody	128
EG_isEquivalent	99	EG_open	51
EG_isIsoPCurve	80	EG_otherCurve	85
EG_isSame	85	EG_quadTess	115
EG_isoCline	79	EG_reall	63
EG_loadModel	53	EG_remakeTess	116
EG_loadTess	136	EG_replaceFaces	103
EG_localToGlobal	134	EG_revision	51
EG_locateTessBody	118	EG_rotate	145
EG_loft	148	EG_ruled	149
EG_makeAttrEFaces	152	EG_saveModel	54
EG_makeEFace	153	EG_saveTess	136
EG_makeFace	92	EG_setFullAttrs	68
EG_makeGeometry	73	EG_setOutLevel	61
EG_makeLoop	93	EG_setTessEdge	130
EG_makeNmWireBody	95	EG_setTessFace	131
EG_makeQuads	122	EG_setTolerance	106
EG_makeSolidBody	94	EG_sewFaces	102
EG_makeTessBody	114	EG_skinning	75
EG_makeTessGeom	112	EG_solidBoolean	137
EG_makeTopology	89	EG_statusTessBody	129
EG_makeTransform	56	EG_strdup	63
EG_mapBody	109	EG_sweep	147
EG_mapBody2	110	EG_tessMassProperties	135
EG_mapTessBody	117	EG_tolerance	106
		EG_updateThread	56