

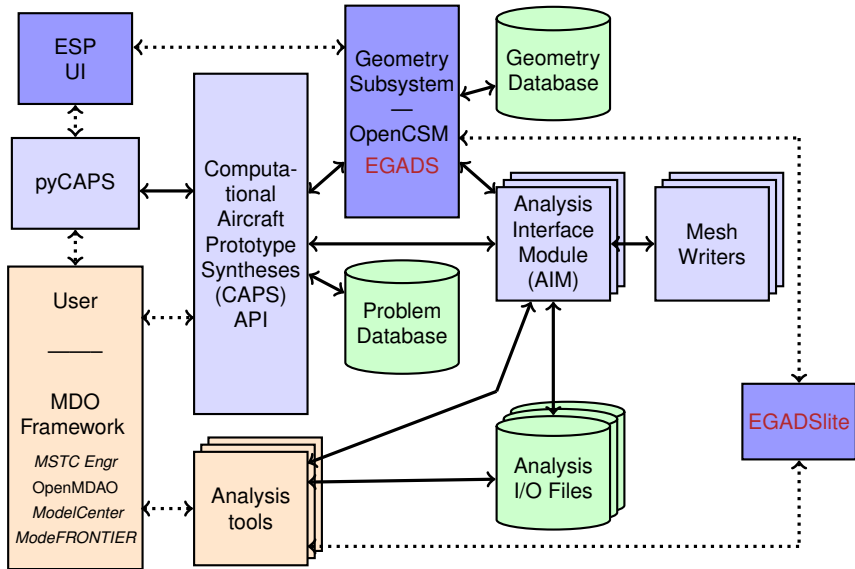


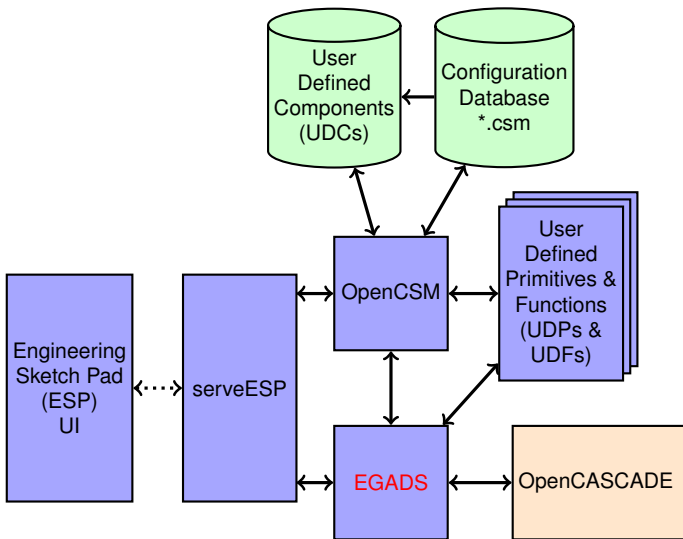
# The Use of Geometry from within the Engineering Sketch Pad – Rev 1.28 The EGADS API

Bob Haimes

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

Geocentric Technologies LLC





The Engineering Geometry Aircraft Design System (EGADS) is an open-source geometry interface to OpenCASCADE

- reduces OpenCASCADE's 17,000 methods to about 100 calls
- supports programming in C, C++, FORTRAN, Python and Julia
- allow *bottom-up* construction via geometric and topological primitives
- allows *top-down* construction via solid creation and Boolean operations
- provides persistent user-defined attributes on topological entities
- adjustable tessellator (vs a surface mesher) with support for finite-differencing in the calculation of parametric sensitivities

## EGADSlite – for HPC Environments

- No construction supported
- Same API and Object model as EGADS
  - Can use EGADS to prototype/build EGADSlite code
- Suitable for an MPI setup:
  - Data export from EGADS via a *stream*
  - Data import to EGADSlite from the *stream*
  - *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

## System Support

- MacOS (Intel or Mx) with **clang**, **ifort/ifx** and/or **gfortran**
- LINUX with **gcc**, **ifort/ifx** and/or **gfortran**
- Windows with Microsoft Visual Studio C++ and **ifort/ifx**
- No globals (but not entirely thread-safe due to OpenCASCADE)
- Various levels of output (0-none, through 3-debug)
- Written in C and C++
- pyEGADS only requires a current version of Python

## EGADS Objects (**egos**)

- Pointer to a C structure – allows for an *Object-based* API
- Treated as “blind” pointers (i.e., not meant to be dereferenced)
- **egos** are INTEGER\*8 variables in FORTRAN

- 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

See [\\$ESP\\_ROOT/doc/EGADS/egads.pdf](#) for a detailed description of all of the functions.

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

See [\\$ESP\\_ROOT/include/egadsErrors.h](#) for a list of the return code **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



## Deleting Objects

- Use the function `EG_deleteObject` to delete Objects
- The Object must be reference *free* – i.e. not used by another
  - Delete in the opposite order of creation
  - If in a Body, delete the Body instead (unless the Body is in a Model)
- `EG_deleteObject` on a Context does not delete the Context
  - Deletes all Objects in the Context that are not in a Body
  - Use `EG_close` to delete all objects in a Context (and the Context)

## Another Rule

- A Body can only be in one Model
  - Copy the Body of interest, then include the copy in the new Model

## surface

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

## 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  $[u, v] = h(t)$

## curve

- 3D curve in the space of 1 running parameter:  $t$
- Types: Line, Circle, Ellipse, Parabola, Hyperbola, Trimmed, Bezier, BSpline, Offset
- All types abstracted to  $[x, y, z] = g(t)$

- All EGADS C/C++ Functions begin with “EG\_”
- There is an attempt to have a descriptive function name
- Inputs are usually at the beginning of the argument list
- Outputs are usually at the end
- Return Values (*icode*):
  - (Almost) all EGADS function have an *icode* return value
  - A value of 0 (EGADS\_SUCCESS) indicates success
  - A negative value indicates an error
    - see `$ESP_ROOT/include/egadsErrors.h` for a list of the return code *defines*.
  - Some functions have a positive return code to indicate partial success or provide other information to the caller

## Create a Geometry Object

```
icode = EG_makeGeometry(ego context, int oclass, int mtype, ego rGeom,  
                        const int *ints, const double *reals,  
                        ego *nGeom);
```

**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:

- 1 **ints** is required for either **mtype** = BEZIER or BSPLINE
- 2 See pages 16-29 of [\\$ESP\\_ROOT/doc/EGADS/egads.pdf](#) for **oclass/mtype** data requirements

## Evaluating the Object

```
icode = EG_evaluate(ego object, double *params, double *result);
```

**object** the input Object

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

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

length $\Rightarrow$	Node – 3	PCurve – 6	Edge Curve – 9	Face 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

Note: You cannot evaluate a DEGENERATE Edge.

## Inverse evaluation on the Object

```
icode = EG_invEvaluate(ego object, double *pos, double *params,  
                      double *result);
```

**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 or EDGE the one value is  $t$   
for SURFACE or FACE 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.

## Boundary Representation – BRep

<p><i>Top</i> <i>Down</i></p> <p>↓</p> <p>↑</p> <p><i>Bottom</i> <i>Up</i></p>	Topology	Geometric Entity	Function
	Model		
	Body	Solid, Sheet, Face, Wire	
	Shell		
	Face	<b>surface</b>	$(x, y, z) = \mathbf{f}(u, v)$
	Loop	<b>pcurve</b> (non-planar)	
	Edge	<b>curve</b>	$(x, y, z) = \mathbf{g}(t)$
	Node	<b>point</b>	

- Nodes that bound Edges may not be exactly on the underlying curves
- Edges in the Loops that trim the Face may not exactly sit on the surface, hence the use of pcurves

## Node

- Contains  $[x, y, z]$
- Types: none

## Edge

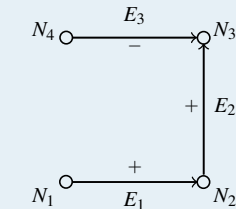
- Has a 3D curve (if not Degenerate)
- Has a  $t$  range ( $t_{min}$  to  $t_{max}$ , where  $t_{min} < t_{max}$ )  
Note: The positive orientation is going from  $t_{min}$  to  $t_{max}$
- Has a Node for  $t_{min}$  and for  $t_{max}$  – can be the same Node
- Types:
  - OneNode – periodic
  - TwoNode – normal
  - Degenerate – single Node,  $t$  range used for the associated pcurve



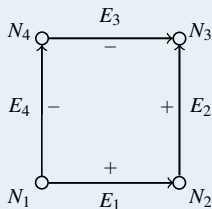


## Loop – without a reference surface

- 1 Free standing connected Edges that can be used in a non-manifold setting (for example in WireBodies)
- 2 A list of connected Edges associated with a Plane (which does not require pcurves)
  - An ordered collection of Edge objects with associated senses
  - Edges must not be Degenerate
  - Types:



Open:  $+E_1 +E_2 -E_3$



Closed:  $+E_1 +E_2 -E_3 -E_4$

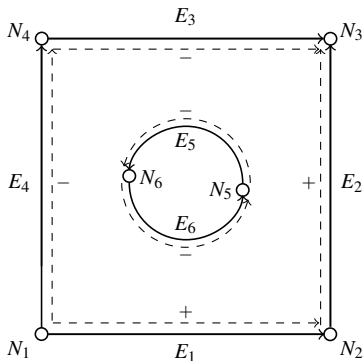
## Loop – with a reference surface

- ① Collections of Edges followed by a corresponding collection of pcurves that define the  $[u, v]$  trimming on the surface
- An ordered collection of Edge objects with associated senses
- 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)
- Trims the surface by maintaining material to the left of the running Loop
- An Edge may be found in a Loop twice (with opposite senses) and with different pcurves.
- Types: Open or Closed (comes back on itself)

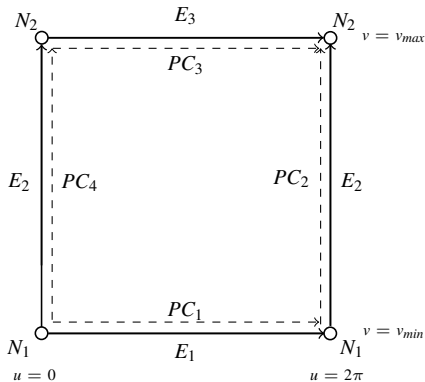
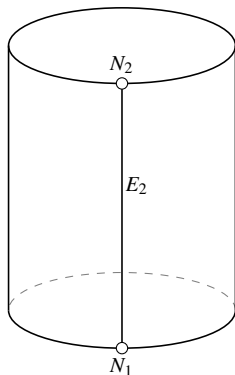
## Face

- A surface bounded by one or more Loops with associated senses
- Only one outer Loop (sense = 1) and any number of inner Loops (sense = -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
- Type is the orientation of the Face based on surface's  $\vec{U} \times \vec{V}$ :
  - 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)



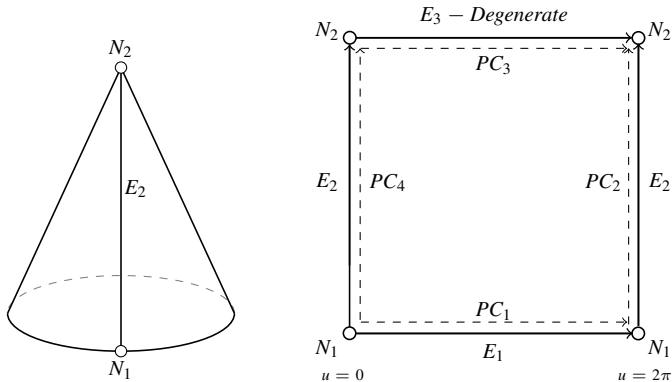
- Outer Loop – right handed/counterclockwise:  $+E_1 +E_2 -E_3 -E_4$
- Inner Loop – left handed/clockwise:  $-E_5 -E_6$



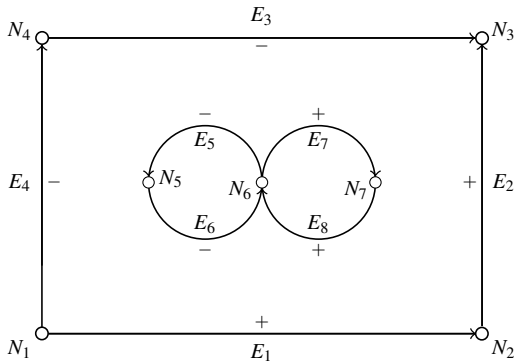
Unrolled periodic cylinder Face

Single Outer Loop – right handed/counterclockwise:

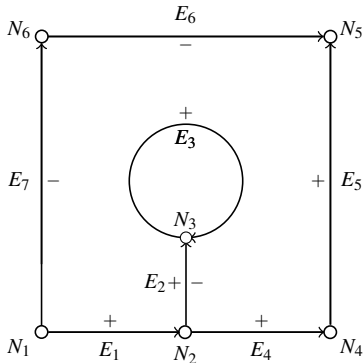
$$+E_1 +E_2 -E_3 -E_2$$



Unrolled Cone



- Outer Loop – right handed/counterclockwise:  $+E_1 +E_2 -E_3 -E_4$
- Inner Loop #1 – left handed/clockwise:  $-E_5 -E_6$
- Inner Loop #2 – left handed/clockwise:  $+E_7 +E_8$



Single Outer Loop – right handed/counterclockwise:

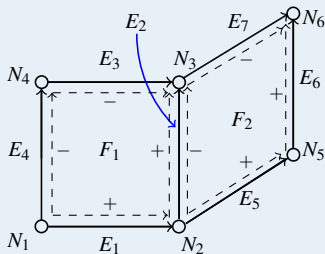
$$+E_1 +E_2 +E_3 -E_2 +E_4 +E_5 -E_6 -E_7$$

Note: PCurve the same for both sides of  $E_2$



## 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
- Types: Open (including non-manifold) or Closed

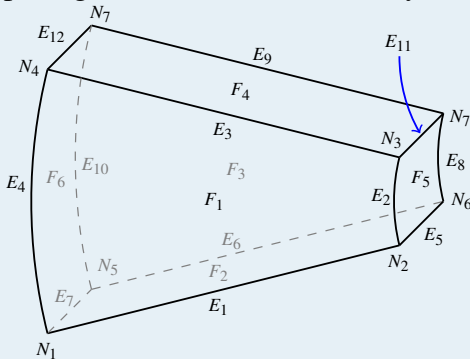


Face #1 Loop:  $+E_1 +E_2 -E_3 -E_4$

Face #2 Loop:  $+E_5 +E_6 -E_7 -E_2$

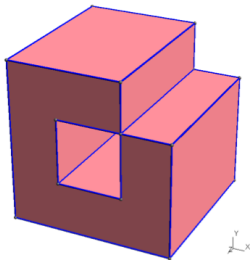
## SolidBody

- Manifold collection of one or more Closed Shells
- One outer Shell (sense = 1); any number of inner (sense = -1)
- Edges (except Degenerate) are found exactly twice (sense =  $\pm 1$ )

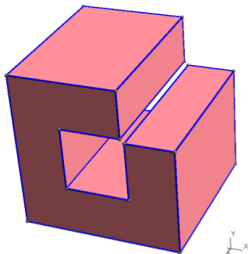


Simple SolidBody: 8 Nodes, 12 Edges, 6 Loops and 6 Faces

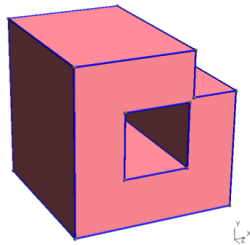
## Manifold vs. Nonmanifold



nonmanifold



manifold



manifold

figure stolen from "An introduction to Geometrical Modelling and Mesh Generation: The Gmsh Companion" by Christophe Geuzaine, Emilie Marchandise & Jean-François Remacle – used without permission!

*Can the geometry be manufactured?*

## Body – including SolidBody

- Container used to aggregate Topology
- Connected to support non-manifold collections at the Model level
- *Owns* all the Objects contained within it
- Types:
  - A WireBody contains a single Loop
  - A FaceBody contains a single Face – IGES import
  - A SheetBody contains a single Shell which can be either non-manifold or manifold (though usually a manifold Body of this type is promoted to a SolidBody)

## Model

- A collection of Bodies – becomes the *Owner* of contained Objects
- Returned by SBO & Sew Functions
- Read and Written by EGADS

## Create a simple Solid Body

```
icode = EG_makeSolidBody(ego context, int stype, const double *data,  
                        ego *body);
```

**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

## Create a Topology Object

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

**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**)

**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

**icode** the integer return code

## Query a Topology Object

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

- topo** the Topology or *Effective Topology* Object to query
- geom** the returned reference Geometry Object (can be **NULL**)
- oclass** the returned Topology Object Class
- mtype** the returned Member Type (depends on **oclass**)
- reals** the real data (at most 4 **doubles** 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  $2 * \text{nchild}$  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

## Queries the Objects in a Body

```
icode = EG_getBodyTopos (const ego body, ego ref, int oclass,  
                        int *ntopo, ego **topos);
```

**body** the Body 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

```
index = EG_indexBodyTopo(const ego body, const ego obj);
```

**body** the Body Object

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

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

## Get the Object in a Body by index

```
icode = EG_objectBodyTopo(const ego body, int oclass, int index,  
                           ego *obj);
```

**body** the Body Object

**oclass** the Topology Object class

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

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

**icode** the integer return code

## Return the Bounding Box info

```
icode = EG_getBoundingBox(const ego object, double *bbox);
```

**object** any topological object

**bbox** 6 **double** 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

```
icode = EG_getMassProperties(const ego object, double *props);
```

**object** can be EDGE, LOOP, FACE, SHELL, BODY or *Effective Topology* counterpart

**props** 14 **double**s 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.

## Memory Functions

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.

```
EG_free(void *ptr);
```

```
void *ptr = EG_alloc(size_t nbytes);
```

```
void *ptr = EG_calloc(size_t nele, size_t size);
```

```
void *ptr = EG_reall(void *pointer, size_t nbytes);
```

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

## Get revision

```
EG_revision(int *imajor, int *iminor, const char **OCCrev);
```

**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

```
icode = EG_open(ego *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

```
icode = EG_close(ego context);
```

**context** the Context Object to close

**icode** the integer return code

Cleans up and closes the Context.

## Delete Object

```
icode = EG_deleteObject(ego 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 or Model Objects.

## Read Geometric data from a File

```
icode = EG_loadModel(ego context, int bitFlag, const char *name,  
                    ego *model);
```

**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.

## Writes the Model to a File

```
icode = EG_saveModel(const ego object, const char *name);
```

**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.

Note: **object** can be a single Body for convenience

## Copy and optionally Transform an Object

```
icode = EG_copyObject(const ego object, void *other, ego *newObj);
```

- object** the Object to copy
- other** Transformation Object, 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.

## Get the Context

```
icode = EG_getContext(ego object, ego *context);
```

- object** the queried Object
- context** the returned owning Context
- icode** the integer return code



- Attributes – metadata consisting of name/value pairs
  - Unique name – no spaces
  - A single type: Integer, Real, String, CSys, Pointer (not persistent)
  - A length (for Integers & Reals)
- Objects
  - Any (non-internal) Object can have multiple Attributes
  - Only Attributes on Topological Objects are copied and are persistent (saved)
- SBO & Intersection Functions
  - Unmodified Topological Objects maintain their Attributes
  - Face Attributes are carried through to the resultant fragments
  - All other Attributes may be lost
- CSys Attributes are modified through Transformations

## 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);
```

- object** the Object to attribute
- name** the name of the attribute
- type** the attribute type:  
ATTRINT, ATTRREAL, ATTRSTRING, ATTRCSYS 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:

- 1 Only the one appropriate attribute value (of **ints**, **reals** or **string**) is required.
- 2 If the **name** already exists the type and value(s) are overwritten.

## Delete an Attribute from an Object

```
icode = EG_attributeDel(ego object, const char *name);
```

**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** then all attributes are removed from this Object.

## The number of Object Attributes

```
icode = EG_attributeNum(ego object, int *nAttr);
```

**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

```
icode = EG_attributeRet(ego object, const char *name, int *type,  
                        int *len, const int **ints,  
                        const double **reals, const char **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:

- 1 Only the appropriate attribute value (of **ints**, **reals** or **string**) is returned.
- 2 The CSys (12 reals) is returned in **reals** after the **len** values.

## Get an Attribute on an Object

```
icode = EG_attributeGet(ego object, int index, const char **name,  
                        int *type, int *len, const int **ints,  
                        const double **reals, const char **string);
```

**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:

- 1 Only the appropriate attribute value (of **ints**, **reals** or **string**) is returned.
- 2 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);
```

**src** the source Object

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

**icode** the integer return code

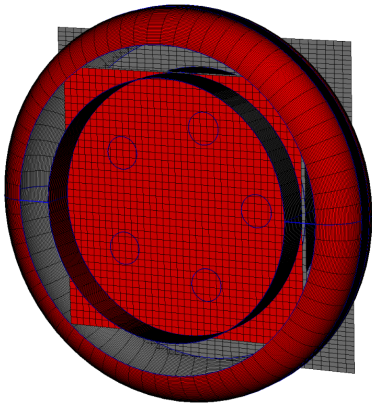
Deletes an attribute from the destination Object and then copies the source's attributes to the destination. ATTRPTR attributes copy the pointer, other types allocate new data and copy the contents of the source.

## Geometry

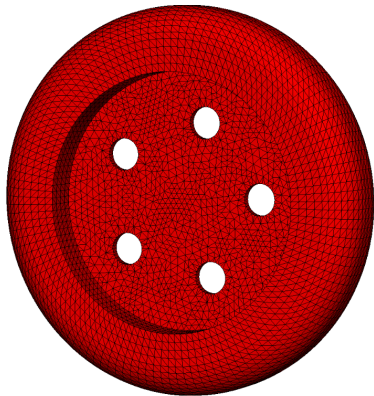
- Unconnected discretization of a range of the Object
  - Polyline for curves at constant  $t$  increments
  - Regular grid for surfaces at constant increments (isoclines)

## Body Topology

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



from `$ESP_ROOT/bin/vGeom`



from `$ESP_ROOT/bin/vTess`



## Creates a Discrete Object from a Body

```
icode = EG_makeTessBody(ego body, double *parms, ego *tess);
```

- 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:
  - parms[0]** – 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.
  - parms[1]** – 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.
  - parms[2]** – 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

See the next page for attribute-based tessellation control.

## Tessellation control at the Topological level

- .tParams** this attribute can be placed on the Body, individual Faces or Edges which overrides **parms** locally (the minimums are used). This attribute must be ATTRREAL and have 3 values (as described in EG\_makeTessBody).
- .tParam** like the attribute **.tParams**, this attribute completely overrides **parms** locally (without using the minimum).
- .tPos** this ATTRREAL attribute on an Edge directly sets the *ts* for interior Edge positions.
- .rPos** this ATTRREAL attribute sets the relative spacing (in arc-length) for interior Edge positions.
- .nPos** this ATTRINT attribute sets the number of interior vertices (length is 1). The spacing is set equal in arc-length.
- .inserts** this ATTRREAL attribute (on a Face) specifies that these vertex  $[u, v]$  positions will be inserted into the tessellation. The length must be 2 times the number of inserts.
- .insert!** like the attribute **.inserts**, this specifies the  $[u, v]$  positions to be inserted, but after these inserts the Face tessellation terminates (i.e., no additional insertions are performed by the normal algorithm).

Note:

An ATTRINT attribute **.tPos** or **.rPos** of length 1 and containing a zero indicates no interior points.

## Gets the Edge discretization data

```
icode = EG_getTessEdge(const ego tess, int eIndex, int *len,  
                      const double **xyzs, const double **ts);
```

- 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` and/or `EG_indexBodyTopo`. A minus index refers to the use of a mapped (+) Edge index from applying the functions `EG_mapBody` and `EG_mapTessBody`.
- len** the returned number of vertices in the Edge discretization
- xyzs** the returned pointer to the set of coordinate data –  $3*\text{len}$  in length
- ts** the returned pointer to the parameter values associated with each vertex –  $\text{len}$  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

```
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);
```

- tess** the input Body Tessellation Object
- fIndex** the Face index (1 bias) – Minus index refers to a mapped (+) Face index (if it exists).
- 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  
orientation consistent with the Face's **mtype**
- 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
- icode** the integer return code

## Status of a Tessellation Object

```
icode = EG_statusTessBody(ego tess, ego *body, int *stat, int *npts);
```

- 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

## Global Lookup

```
icode = EG_localToGlobal(const ego tess, int ind, int locl, int *gbl);
```

**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

**icode** the integer return code

## Gets the vertex type and index

```
icode = EG_getGlobal(const ego tess, int global, int *pytpe,  
                    int *pindex, double *xyz);
```

**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**)

**icode** the integer return code

In the *exercise/session02* directory:

- Examine the Makefile (or NMakefile on Windows). Notice the library(s) included.
- Build the executable and run it. The output should look like:

```
Using EGADS 1.28 Interim Release with OpenCASCADE 7.8.1
```

```
Number of Bodies = 2
```

```
Body 0: Name = capsLength String = cm
```

```
  Tessellation 0 npts = 1306 (Solid)
```

```
  Volumes = 7.280172e+00 7.296560e+00
```

```
Body 1: Name = capsLength String = cm
```

```
  Tessellation 0 npts = 2263 (Solid)
```

```
  Volumes = 3.454432e+01 3.464152e+01
```

```
EGADS Info: 0 Objects, 0 Reference in Use (of 247) at Close!
```

- Modify `myExample.c` to print all attributes using `EG_getBodyTopos` for all of the `FACES`, `EDGES` and `NODEs` in each `Body`. Note that you must *free* the vectors of `Objects`.
- Modify `myExample.c` to traverse the `BRep Topology` from `Model` to `Nodes` using `EG_getTopology` and output all attributes attached to each `Topological entity`.