Engineering Sketch Pad

Writing a UDP or UDF
at Revision 1.18

Bob Haimes
Massachusetts Institute of Technology

John F. Dannenhoffer, III
Syracuse University

Overview

- Objects
- The API
  - Base & Utility Functions
  - Attribution
  - Geometry
  - Geometry / Topology
  - Topology
  - Tessellation
  - Top-Down Construction
- Steps to writing a UDP/UDF
- Tire UDP
  - structure of code
  - code walk-through
  - stand-alone execution
  - execution as a UDP

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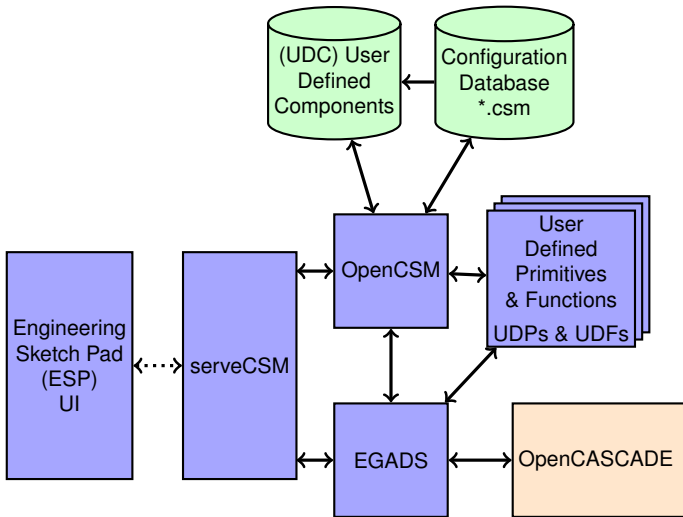
The Engineering Geometry Aircraft Design System (EGADS) is an open-source geometry interface to OpenCASCADE

- reduces OpenCASCADE's 17,000 methods to about 70 calls
  - Supports C, C++ & FORTRAN
- provides *bottom-up* and/or *top-down* construction
- geometric primitives
  - curve: line, circle, ellipse, parabola, hyperbola, offset, bezier, BSpline (including NURBS)
  - surface: plane, spherical, conical, cylindrical, toroidal, revolution, extrusion, offset, bezier, BSpline (including NURBS)
- solid creation and Boolean operations (*top-down*)
- 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

EGADS Overview

- System Support
  - 64-bit pointers (only)
  - Mac OSX with **clang**, **ifort** and/or **gfortran**
  - LINUX with **gcc**, **ifort** and/or **gfortran**
  - Windows with Microsoft Visual Studio C++ and **ifort**
  - No globals (but not thread-safe due to OpenCASCADE)
  - Various levels of output (0-none, through 3-debug)
  - Written in C and C++

- EGADS Objects (**ego**s)
  - Pointer to a C structure – allows for an *Object-based* API
  - Treated as "blind" pointers (i.e., not meant to be dereferenced)
  - **ego**s 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

EGADS Objects – Attribution

- Attributes – metadata consisting of name/value pairs
  - Unique name – no spaces
  - A single type: Integer, Real, String, CSys
  - A length (not for strings)
- 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 are lost
- CSys Attributes are modified through Transformations

`EGADS` Geometry Objects

- surface
  - 3D surfaces of 2 parameters $[u, v]$
  - Types: Plane, Spherical, Cylindrical, Revolution, Toroidal, 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 $[u, v] = g(t)$
- curve
  - 3D curve – single running parameter (t)
  - Same types as pcurve but abstracted to $[x, y, z] = g(t)$

Boundary Representation – BRep

Top Down

Bottom Up

Topological Entity	Geometric Entity	Function
Model		
Body	Solid, Sheet, Wire	
Shell		
Face	**surface**	$(x, y, z) = \mathbf{f}(u, v)$
Loop		
Edge	**curve**	$(x, y, z) = \mathbf{g}(t)$
Node	**point**	

- Nodes that bound Edges may not be on underlying curves
- Edges in the Loops that trim the Face may not 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 that define the connected *Wire/Contour/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
  - Types: Open or Closed (comes back on itself)

- Loop – with a reference surface
  1. Collections of Edges (like 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 = 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 $U \otimes V$:
    - 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
  - Types: Open (including non-manifold) or Closed
- SolidBody
  - A manifold collection of one or more Closed Shells with associated senses
  - There may be only one outer Shell (sense = 1) and any number of inner Shells (sense = -1)
  - Edges (except Degenerate) are found exactly twice (sense = ± 1)

- Body – including SolidBody
  - Container used to aggregate Topology
  - Connected to support non-manifold collections at the Model level
  - *Owns* all the Objects contained within
  - 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`

- Helper Functions
    - makeLoop
        - Connects unrelated (via Nodes) Edges from a list
        - Uses input tolerance to match entities
        - Result may be multiple Loops
   - makeFace
        - From Closed Planar Loop
        - From surface with limits
   - sewFaces
        - Connects Faces with unrelated Topology
        - Uses input tolerance to match entities
        - Returns a Model – may have multiple Bodies
        - Can connect in a nonmanifold manner

- 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
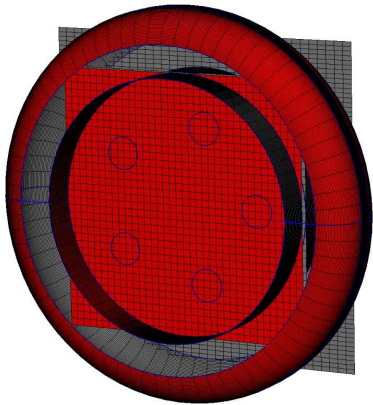
from $ESP_ROOT/bin/vGeom
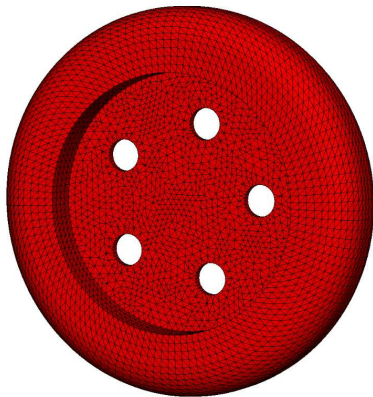
from $ESP_ROOT/bin/vTess

- Function names begin with "EG_" – or –
  "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` needs to be used when marked as "freeable"
- **ego**s have:
  - *Owner*: Context, Body, or Model
  - Reference Objects (objects they depend upon)
- When a Body is made, all included Objects are copied – not referenced

- Deleting Objects
  - Use the function "EG_deleteObject" (or "ig_deleteobject")
  - 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 (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

```
void EG_revision( int *major, int *minor, const char **OCCrev )
int  EG_open( ego *context )
int  EG_loadModel( ego contxt, int bflg, const char *name, ego *model )
int  EG_saveModel( const ego model, const char *name )
int  EG_deleteObject( ego object )
int  EG_makeTransform( ego context, const double *xform, ego *oform )
int  EG_getTransformation( const ego oform, double *xform )
int  EG_getContext( ego object, ego *context )
int  EG_setOutLevel( ego context, int outLevel )
int  EG_getInfo( const ego object, int *oclass, int *mtype,
                 ego *topObj, ego *prev, ego *next )
int  EG_copyObject( const ego obj, /*@null@*/ void *oform, ego *copy )
int  EG_flipObject( const ego object, ego *flippedCopy )
int  EG_close( ego context )

/*@null@*/ /*@out@*/ /*@only@*/ void *EG_alloc( int nbytes )
/*@null@*/ /*@only@*/ void *EG_calloc( int nele, int size )
/*@null@*/ /*@only@*/ void *EG_reall( /*@null@*/ /*@returned@*/
                                      /*@only@*/ void *ptr, int nbytes )
void EG_free(/*@null@*/ /*@only@*/ void *pointer )
```

```
int EG_attributeAdd( ego object, const char *name, int type, int len,
                     /*@null@*/ const int *ints,
                     /*@null@*/ const double *reals,
                     /*@null@*/ const char *str )
int EG_attributeDel( ego object, /*@null@*/ const char *name )
int EG_attributeNum( const ego object, int *num )
int EG_attributeGet( const ego object, int index, const char **name,
                     int *atype, int *len,
                     /*@null@*/ const int **ints,
                     /*@null@*/ const double **reals,
                     /*@null@*/ const char **str )
int EG_attributeRet( const ego object, const char *name, int *atype,
                     int *len,
                     /*@null@*/ const int **ints,
                     /*@null@*/ const double **reals,
                     /*@null@*/ const char **str )
int EG_attributeDup( const ego srcObj, ego dstObj )
```

```
int EG_makeGeometry( ego context, int oclass, int mtype,
                     /*@null@*/ ego ref, /*@null@*/ const int *ivec,
                     const double *rvec, ego *geom )
int EG_getGeometry( const ego geom, int *oclass, int *mtype,
                     ego *refGeom, int **ivec, double **rvec )

int EG_approximate( ego context, int maxdeg, double tol,
                     const int *sizes, const double *xyzs, ego *bsplin )
int EG_fitTriangles( ego context, int npts, double *xyzs, int ntris,
                     const int *tris, /*@null@*/ const int *tric,
                     double tol, ego *bspline )
int EG_otherCurve( const ego surface, const ego curve, double tol,
                     ego *newcurve )
int EG_isoCline( const ego surface, int UV, double val, ego *newcurve )
```

```
int EG_getRange( const ego obj, double *range, int *periodic )
int EG_getArea( ego obj, /*@null@*/ const double *limits,
                double *area )
int EG_evaluate( const ego obj, const double *param, double *results )
int EG_invEvaluate( const ego obj, double *xyz, double *param,
                    double *results )
int EG_invEvaluateGuess( const ego obj, double *xyz, double *param,
                         double *results )
int EG_arcLength( const ego obj, double t1, double t2, double *alen )
int EG_curvature( const ego obj, const double *parm, double *results )
int EG_isSame( const ego obj1, const ego obj2 )
int EG_convertToBSpline( ego obj, ego *bspline )
```

```
int EG_makeTopology( ego context, /*@null@*/ ego geom, int oclass,
                     int mtype, /*@null@*/ double *limits,
                     int nChildren, /*@null@*/ ego *children,
                     /*@null@*/ int *senses, ego *topo )
int EG_makeSolidBody( ego context, int stype, const double *rvec,
                      ego *body )
int EG_getTopology( const ego topo, ego *geom, int *oclass, int *type,
                    /*@null@*/ double *limits, int *nChildren,
                    ego **children, int **senses )

int EG_makeLoop( int nedge, ego *edges, /*@null@*/ ego geom,
                 double toler, ego *result )
int EG_makeFace( ego object, int mtype,
                 /*@null@*/ const double *limits, ego *face )
int EG_sewFaces( int nobj, const ego *faces, double toler, int flag,
                 ego *model )

int EG_getEdgeUV( const ego face, const ego edge, int sense, double t,
                  double *UV )
int EG_getBoundingBox( const ego topo, double *bbox )
int EG_getMassProperties( const ego topo, double *result )
int EG_isEquivalent( const ego topo1, const ego topo2 )
```

```
int EG_getBody( const ego topo, ego *body )
int EG_getBodyTopos( const ego body, /*@null@*/ ego src, int oclass,
                     int *ntopo, ego **topos )
int EG_indexBodyTopo( const ego body, const ego src )
int EG_inTopology( const ego topo, const double *xyz )
int EG_inFace( const ego face, const double *uv )

int EG_getTolerance( const ego topo, double *tol )
int EG_setTolerance( ego topo, double tol )

int EG_replaceFaces( const ego body, int nobj, ego *objs, ego *result )
int EG_mapBody( const ego sBody, const ego dBody, const char *fAttr,
                ego *mapBody )
int EG_matchBodyFaces( const ego body1, const ego body2, double toler,
                       int *nmatch, int **match )
```

```
int EG_makeTessGeom( ego obj, double *params, int *sizes, ego *tess )
int EG_getTessGeom( const ego tess, int *sizes, double **xyz )

int EG_makeTessBody( ego body, double *params, ego *tess )
int EG_mapTessBody( ego tess, ego body, ego *mapTess )
int EG_getTessEdge( const ego tess, int eIndex, int *len,
                    const double **xyz, const double **t )
int EG_getTessFace( const ego tess, int fIndex, int *len,
                    const double **xyz, const double **uv,
                    const int **ptype, const int **pindex,
                    int *ntri, const int **tris, const int **tric )
int EG_statusTessBody( ego tess, ego *body, int *state, int *nGlobal )
int EG_localToGlobal( const ego tess, int index, int local,
                      int *global )
int EG_getGlobal( const ego tess, int global, int *ptype, int *pindex,
                  /*@null@*/ double *xyz )
```

Note: All numbering in EGADS is 1-biased

ptype – the point type: (-) Face local index, (0) Node, (+) Edge local index
pindex – the point topological index (1 bias)

```
int EG_solidBoolean( const ego src, const ego tool, int oper,
                     ego *model )
int EG_intersection( const ego src, const ego tool, int *nedge,
                     /*@null@*/ ego **facEdg, ego *model )
int EG_imprintBody( const ego src, int nedge,
                    const ego *facEdg, ego *result )
int EG_filletBody( const ego src, int nedge, const ego *edges,
                   double radius, ego *result,
                   /*@null@*/ int **facemap )
int EG_chamferBody( const ego src, int nedge, const ego *edges,
                    const ego *faces, double dis1, double dis2,
                    ego *result, /*@null@*/ int **facemap )
int EG_hollowBody( const ego src, int nface, const ego *faces,
                   double offset, int join, ego *result,
                   /*@null@*/ int **facemap )
int EG_extrude( const ego src, double dist, const double *dir,
                ego *result )
int EG_rotate( const ego src, double angle, const double *axis,
               ego *result )
int EG_sweep( const ego src, const ego spine, int mode, ego *result )
int EG_blend(int nsec, const ego *secs, /*@null@*/ double *rc1,
             /*@null@*/ double *rcN, ego *result )
int EG_ruled( int nsec, const ego *secs, ego *result )
```

- **Draw a picture**
- Define input and output parameters
    - name (case-insensitive)
    - type (ATTRSTRING, ATTRINT, -ATTRINT, ATTRREAL, -ATTRREAL, ATTRREALSEN)
    - size
    - default value(s)
- Build the Body (stand-alone)
    - bottom-up
    - top-down
    - combination
- Test stand-alone with vTess
- Write a .csm file
- Test the UDP/UDF

```
#define NUMUDPARGS num          /* number of inputs and outputs */
#define NUMUDPINPUTBODYS num    /* number of input bodies - UDF only */


int
udpExecute(ego object,          /* (in)  EGADS Context -- UDP
                                          EGADS Model   -- UDF */
           ego *ebody,          /* (out) Body pointer */
           int *nMesh,          /* (out) number of associated meshes */
           char *string[])      /* (out) error message */


int
udpSensitivity(ego ebody,       /* (in)  Body pointer */
               int npnt,        /* (in)  number of points */
               int entType,     /* (in)  OCSM entity type */
               int entIndex,    /* (in)  OCSM entity index (bias-1) */
               double uvs[],    /* (in)  parametric coords for eval */
               double vels[])   /* (out) velocities */
```
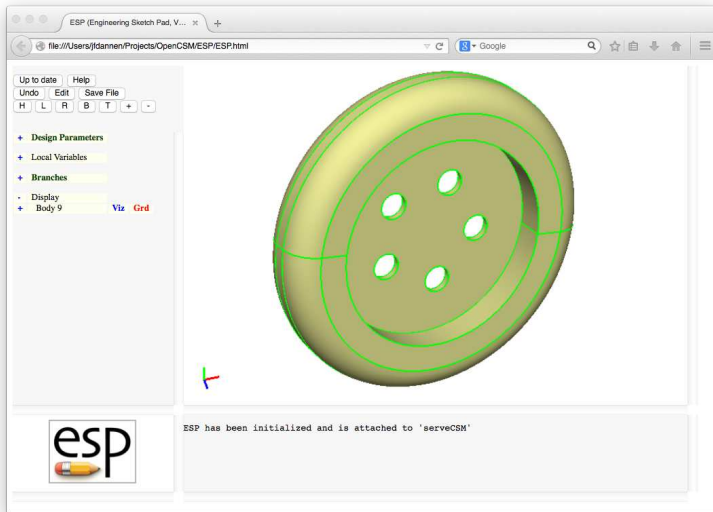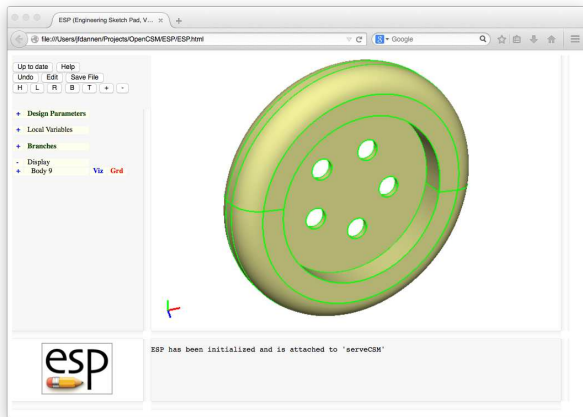
- cd to `$ESP_ROOT/doc/UDP_UDF/data`
- Source file is `udpTire.c`
- To build under LINUX/OSX
    - `make`
- To build under Windows
    - `nmake -f NMakefile`
- To run stand-alone
    - `tire`
    - `$ESP_ROOT/bin/vTess tire.egads`
        - open browser on `$ESP_ROOT/bin/wv.html`
- To run in `ESP`
    - `$ESP_ROOT/bin/serveCSM tire.csm`

Example of both *Bottom Up* and *Top Down* Construction

Name	Description
`width`	width
`minrad`	minimum radius
`maxrad`	maximum radius
`filletrad`	fillet radius at outside
`platethick`	wheel thickness
`bolts`	number of bolt holes
`patternrad`	radius of bolt circle
`boltrad`	radius of bolt holes
`volume`	volume (output)

```
#define NUMUDPARGS 9
#include "udpUtilities.h"

/* shorthands for accessing argument values and velocities */
#ifdef UDP
    #define WIDTH(IUDP)        ((double *) (udps[IUDP].arg[0].val))[0]
    #define MINRAD(IUDP)       ((double *) (udps[IUDP].arg[1].val))[0]
    #define MAXRAD(IUDP)       ((double *) (udps[IUDP].arg[2].val))[0]
    #define FILLETRAD(IUDP)    ((double *) (udps[IUDP].arg[3].val))[0]
    #define PLATETHICK(IUDP)   ((double *) (udps[IUDP].arg[4].val))[0]
    #define PATTERN(IUDP)      ((double *) (udps[IUDP].arg[5].val))[0]
    #define BOLTS(IUDP)        ((double *) (udps[IUDP].arg[6].val))[0]
    #define BOLTRAD(IUDP)      ((double *) (udps[IUDP].arg[7].val))[0]
    #define VOLUME(IUDP)       ((double *) (udps[IUDP].arg[8].val))[0]
    /* data about possible arguments */
    static char   *argNames[NUMUDPARGS] = {"width",  "minrad", "maxrad", "fillrad", "platethick", "p
    static int     argTypes[NUMUDPARGS] = {ATTRREAL, ATTRREAL, ATTRREAL, ATTRREAL, ATTRREAL,    AT
    static int     argIdefs[NUMUDPARGS] = {0,        0,        0,        0,         0,           0,
    static double  argDdefs[NUMUDPARGS] = {0.,       0.,       0.,       0.,        0.,          0.

    #include "udpUtilities.c"
#else
    #define WIDTH(IUDP)        5.0
    #define MINRAD(IUDP)       8.0
    #define MAXRAD(IUDP)       12.0
    #define FILLETRAD(IUDP)    2.0
    #define PLATETHICK(IUDP)   0.5
    #define PATTERN(IUDP)      4.0
    #define BOLTS(IUDP)        5.0
    #define BOLTRAD(IUDP)      1.0
    #define VOLUME(IUDP)       myVolume
 #endif
```

```
int main(int  argc, char *argv[])
{
    int   status, nMesh;
    char *string;
    ego   context, ebody, emodel;

    status = EG_open(&context);
    printf("EG_open -> status=%d\n", status);
    if (status < 0) exit(EXIT_FAILURE);

    /* call the execute routine */
    status = udpExecute(context, &ebody, &nMesh, &string);
    printf("udpExecute -> status=%d\n", status);
    if (status < 0) exit(EXIT_FAILURE);
    EG_free(string);

    /* make and dump the model */
    status = EG_makeTopology(context, NULL, MODEL, 0, NULL,
                             1, &ebody, NULL, &emodel);
    printf("EG_makeTopology -> status=%d\n", status);
    if (status < 0) exit(EXIT_FAILURE);
    status = EG_saveModel(emodel, "tire.egads");
    printf("EG_saveModel -> status=%d\n", status);
    if (status < 0) exit(EXIT_FAILURE);

    /* cleanup */
    status = EG_deleteObject(emodel);
    printf("EG_close -> status=%d\n", status);
    status = EG_close(context);
    printf("EG_close -> status=%d\n", status);
    return EXIT_SUCCESS;
}
```

- Draw a *sketch*, with Nodes, Edges, and Faces numbered
- Define the inputs and outputs
  - check size (scalar vs. multi-valued)
  - check validity
- Build basic tire from bottom up
  - 8 Nodes
  - 8 Edges (linear) at the equator
    - generate a linear curve
    - inverse evaluate at Nodes to get t_{beg} and t_{end}
    - make the Edge
  - 8 Edges (circular)
    - generate the circular curve
    - inverse evaluate at Nodes to get t_{beg} and t_{end}
    - ensure $t_{end} > t_{beg}$ by increasing t_{beg} by 2π if needed
   - make the Edge
  - . . .
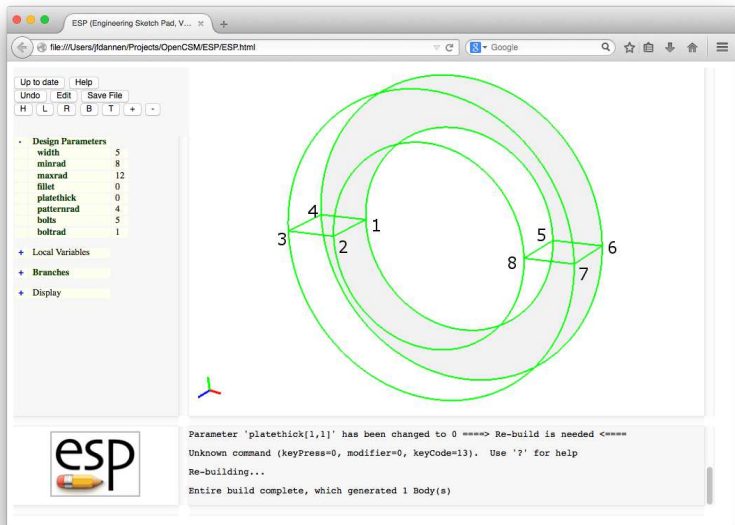
```
/* Node locations */

node1[0] = -MINRAD(0);   node1[1] = 0;   node1[2] = -WIDTH(0) / 2;
node2[0] = -MINRAD(0);   node2[1] = 0;   node2[2] =  WIDTH(0) / 2;
node3[0] = -MAXRAD(0);   node3[1] = 0;   node3[2] =  WIDTH(0) / 2;
node4[0] = -MAXRAD(0);   node4[1] = 0;   node4[2] = -WIDTH(0) / 2;
node5[0] =  MINRAD(0);   node5[1] = 0;   node5[2] = -WIDTH(0) / 2;
node6[0] =  MAXRAD(0);   node6[1] = 0;   node6[2] = -WIDTH(0) / 2;
node7[0] =  MAXRAD(0);   node7[1] = 0;   node7[2] =  WIDTH(0) / 2;
node8[0] =  MINRAD(0);   node8[1] = 0;   node8[2] =  WIDTH(0) / 2;

/* make the Nodes */

status = EG_makeTopology(context, NULL, NODE, 0, node1, 0, NULL, NULL, &enodes[0]);
if (status != EGADS_SUCCESS) goto cleanup;
status = EG_makeTopology(context, NULL, NODE, 0, node2, 0, NULL, NULL, &enodes[1]);
if (status != EGADS_SUCCESS) goto cleanup;
status = EG_makeTopology(context, NULL, NODE, 0, node3, 0, NULL, NULL, &enodes[2]);
if (status != EGADS_SUCCESS) goto cleanup;
status = EG_makeTopology(context, NULL, NODE, 0, node4, 0, NULL, NULL, &enodes[3]);
if (status != EGADS_SUCCESS) goto cleanup;
status = EG_makeTopology(context, NULL, NODE, 0, node5, 0, NULL, NULL, &enodes[4]);
if (status != EGADS_SUCCESS) goto cleanup;
status = EG_makeTopology(context, NULL, NODE, 0, node6, 0, NULL, NULL, &enodes[5]);
if (status != EGADS_SUCCESS) goto cleanup;
status = EG_makeTopology(context, NULL, NODE, 0, node7, 0, NULL, NULL, &enodes[6]);
if (status != EGADS_SUCCESS) goto cleanup;
status = EG_makeTopology(context, NULL, NODE, 0, node8, 0, NULL, NULL, &enodes[7]);
if (status != EGADS\_SUCCESS) goto cleanup;
```
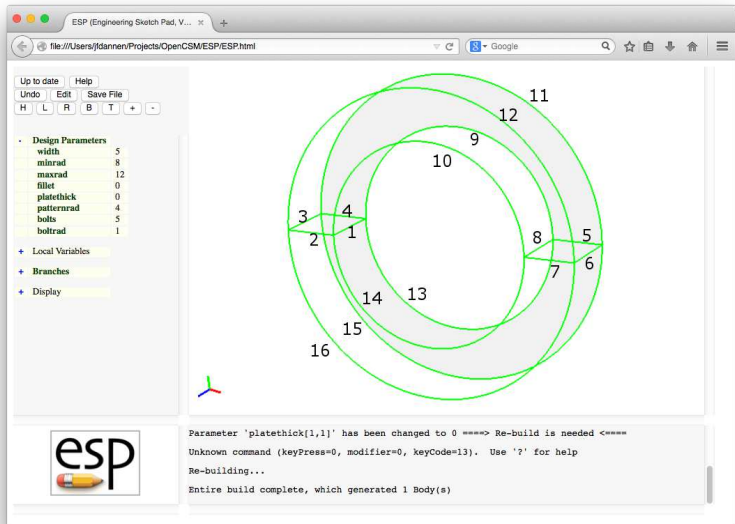
```
/* make (linear) Edge 1 */

data[0] = node1[0];
data[1] = node1[1];
data[2] = node1[2];
data[3] = node2[0] - node1[0];
data[4] = node2[1] - node1[1];
data[5] = node2[2] - node1[2];
status  = EG_makeGeometry(context, CURVE, LINE, NULL, NULL, data, &ecurve[0]);
if (status != EGADS_SUCCESS) goto cleanup;

status = EG_invEvaluate(ecurve[0], node1, &trange[0], data);
if (status != EGADS_SUCCESS) goto cleanup;
status = EG_invEvaluate(ecurve[0], node2, &trange[1], data);
if (status != EGADS_SUCCESS) goto cleanup;

elist[0] = enodes[0];
elist[1] = enodes[1];
status  = EG_makeTopology(context, ecurve[0], EDGE, TWONODE, trange, 2, elist, NULL,
                          &eedges[0]);
if (status != EGADS_SUCCESS) goto cleanup;
```

```
/* data used in creating the arcs */

axis1[0] = 1;   axis1[1] = 0;   axis1[2] = 0;
axis2[0] = 0;   axis2[1] = 1;   axis2[2] = 0;
axis3[0] = 0;   axis3[1] = 0;   axis3[2] = 1;
cent1[0] = 0;   cent1[1] = 0;   cent1[2] = -WIDTH(0) / 2;
cent2[0] = 0;   cent2[1] = 0;   cent2[2] =  WIDTH(0) / 2;

/* make (circular) Edge 9 */

data[0] = cent1[0];   data[1] = cent1[1];   data[2] = cent1[2];
data[3] = axis1[0];   data[4] = axis1[1];   data[5] = axis1[2];
data[6] = axis2[0];   data[7] = axis2[1];   data[8] = axis2[2];   data[9] = MINRAD(0);
status = EG_makeGeometry(context, CURVE, CIRCLE, NULL, NULL, data, &ecurve[8]);
if (status != EGADS_SUCCESS) goto cleanup;


status = EG_invEvaluate(ecurve[8], node5, &trange[0], data);
if (status != EGADS_SUCCESS) goto cleanup;

status = EG_invEvaluate(ecurve[8], node1, &trange[1], data);
if (status != EGADS_SUCCESS) goto cleanup;
if (trange[0] > trange[1]) trange[1] += TWOPI;   /* ensure trange[1] > trange[0] */

elist[0] = enodes[4];
elist[1] = enodes[0];
status   = EG_makeTopology(context, ecurve[8], EDGE, TWONODE, trange, 2, elist, NULL,
                           &eedges[8]);
if (status != EGADS_SUCCESS) goto cleanup;
```
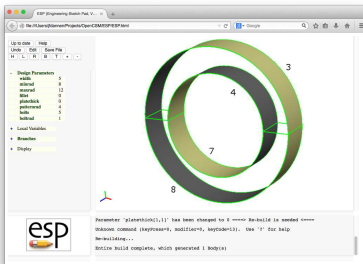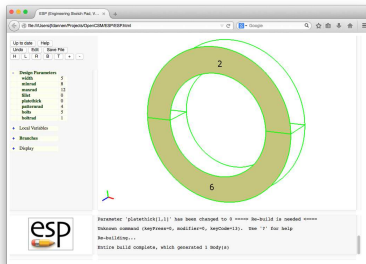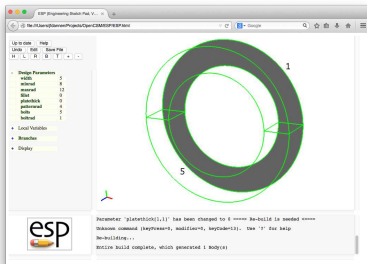
- Continue bottom up build
  - 4 Faces (planar)
    - make a Loop of 4 Edges
    - make the (planar) Face
  - 4 Faces (cylindrical)
    - make cylindrical surface
    - make a PCurve for each Edge that bounds Face
    - make a Loop of 4 Edges and 4 PCurves
    - make the (cylindrical) Face
  - 1 Shell that combines the 8 Faces
  - 1 Solid Body from the Shell

```
/* make the outer cylindrical surface */

data[0] = cent1[0];   data[1] = cent1[1];   data[2]  = cent1[2];
data[3] = axis1[0];   data[4] = axis1[1];   data[5]  = axis1[2];
data[6] = axis2[0];   data[7] = axis2[1];   data[8]  = axis2[2];
data[9] = axis3[0];   data[10] = axis3[1];   data[11] = axis3[2];   data[12] = MAXRAD(0);
status = EG_makeGeometry(context, SURFACE, CYLINDRICAL, NULL, NULL, data, &esurface[0]);
if (status != EGADS_SUCCESS) goto cleanup;

/* make the inner cylindrical surface */

data[0] = cent1[0];   data[1] = cent1[1];   data[2]  = cent1[2];
data[3] = axis1[0];   data[4] = axis1[1];   data[5]  = axis1[2];
data[6] = axis2[0];   data[7] = axis2[1];   data[8]  = axis2[2];
data[9] = axis3[0];   data[10] = axis3[1];   data[11] = axis3[2];   data[12] = MINRAD(0);
status = EG_makeGeometry(context, SURFACE, CYLINDRICAL, NULL, NULL, data, &esurface[1]);
if (status != EGADS_SUCCESS) goto cleanup;

/* make (planar) Face 1 */

sense[0] = SFORWARD;   sense[1] = SREVERSE;   sense[2] = SFORWARD;   sense[3] = SFORWARD;
elist[0] = eedges[3];  elist[1] = eedges[8];  elist[2] = eedges[4];  elist[3] = eedges[10];
status   = EG_makeTopology(context, NULL, LOOP, CLOSED, NULL, 4, elist, sense, &eloop);
if (status != EGADS_SUCCESS) goto cleanup;

status = EG_makeFace(eloop, SFORWARD, NULL, &efaces[0]);
if (status != EGADS_SUCCESS) goto cleanup;
```

```
/* make (cylindrical) Face 3 */
status = EG_otherCurve(esurface[0], ecurve[2], 0, &epcurve[0]);
if (status != EGADS_SUCCESS) goto cleanup;

status = EG_otherCurve(esurface[0], ecurve[10], 0, &epcurve[1]);
if (status != EGADS_SUCCESS) goto cleanup;

status = EG_otherCurve(esurface[0], ecurve[5], 0, &epcurve[2]);
if (status != EGADS_SUCCESS) goto cleanup;

status = EG_otherCurve(esurface[0], ecurve[11], 0, &epcurve[3]);
if (status != EGADS_SUCCESS) goto cleanup;

sense[0] = SFORWARD;   sense[1] = SREVERSE;   sense[2] = SFORWARD;   sense[3] = SFORWARD;
elist[0] = eedges[2];  elist[1] = eedges[10]; elist[2] = eedges[5];  elist[3] = eedges[11];
elist[4] = epcurve[0]; elist[5] = epcurve[1]; elist[6] = epcurve[2]; elist[7] = epcurve[3];
status = EG_makeTopology(context, esurface[0], LOOP, CLOSED, NULL, 4, elist, sense, &eloop);
if (status != EGADS_SUCCESS) goto cleanup;

status = EG_makeTopology(context, esurface[0], FACE, SREVERSE, NULL, 1, &eloop, sense,
                         &efaces[2]);
if (status != EGADS_SUCCESS) goto cleanup;
:
:

 /* make the shell and initial Body */
status = EG_makeTopology(context, NULL, SHELL, CLOSED, NULL, 8, efaces, NULL, &eshell);
if (status != EGADS_SUCCESS) goto cleanup;

status = EG_makeTopology(context, NULL, BODY, SOLIDBODY, NULL, 1, &eshell, NULL, &ebody1);
if (status != EGADS_SUCCESS) goto cleanup;
```
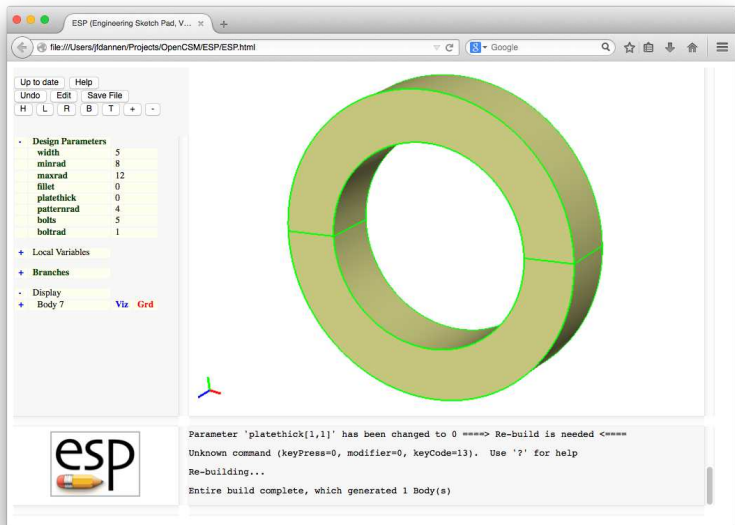
- Modify the Body top-down
  - fillet on outer Edges
    - identify the 4 Edges
  - add wheel
    - cylinder that is "unioned" with the tire
  - add pattern of holes
    - cylinders that are "subtracted" from the wheel
- Compute and return the "output" variables
- Note: this entire UDP could have been written top-down, but was broken up to show the steps needed in bottom-up construction

```
/* add fillets if desired (result is ebody2) */
if (FILLETRAD(0) > 0.0) {
  elist[0] = eedges[10]; elist[1] = eedges[11]; elist[2] = eedges[14]; elist[3] = eedges[15];
  status = EG_filletBody(ebody1, 4, elist, FILLETRAD(0), &ebody2, NULL);
  if (status != EGADS_SUCCESS) goto cleanup;
  status = EG_deleteObject(ebody1);
  if (status != EGADS_SUCCESS) goto cleanup;
} else {
  ebody2 = ebody1;
}
```
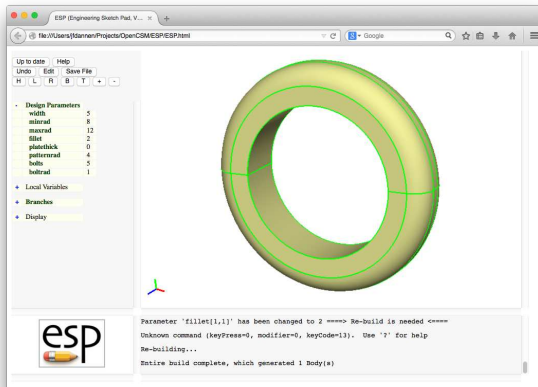
```
if (PLATETHICK(0) > 0.0) {
  data[0] = 0;    data[1] = 0;    data[2] =  PLATETHICK(0) / 2;
  data[3] = 0;    data[4] = 0;    data[5] = -PLATETHICK(0) / 2;
  data[6] = (MINRAD(0) + MAXRAD(0)) / 2;
  status = EG_makeSolidBody(context, CYLINDER, data, &ebody3);
  if (status != EGADS_SUCCESS) goto cleanup;

  status = EG_solidBoolean(ebody2, ebody3, FUSION, &emodel);
  if (status != EGADS_SUCCESS) goto cleanup;

  status = EG_deleteObject(ebody2);
  if (status != EGADS_SUCCESS) goto cleanup;
  status = EG_deleteObject(ebody3);
  if (status != EGADS_SUCCESS) goto cleanup;
  status = EG_getTopology(emodel, &eref, &oclass, &mtype, data, &nchild, &echilds, &senses);
  if (status != EGADS_SUCCESS) goto cleanup;

  if (oclass != MODEL || nchild != 1) {
    printf("No model or are returning more than one body ochild = %d, nchild = %d/n",
           oclass, nchild);
    status = -999;
    goto cleanup;
  }

  status = EG_copyObject(echilds[0], NULL, &source);
  if (status != EGADS_SUCCESS) goto cleanup;
  status = EG_deleteObject(emodel);
  if (status != EGADS_SUCCESS) goto cleanup;
```

```
/* add bolt holes */
for (i = 0; i < NINT(BOLTS(0)); i++) {
    data[0] = PATTERN(0) * cos(i * (2 * PI / BOLTS(0)));
    data[1] = PATTERN(0) * sin(i * (2 * PI / BOLTS(0)));
    data[2] = PLATETHICK(0) / 2;
    data[3] = PATTERN(0) * cos(i * (2 * PI / BOLTS(0)));
    data[4] = PATTERN(0) * sin(i * (2 * PI / BOLTS(0)));
    data[5] = -PLATETHICK(0) / 2;
    data[6] = BOLTRAD(0);

    status = EG_makeSolidBody(context, CYLINDER, data, &ebody4);
    if (status != EGADS_SUCCESS) goto cleanup;

    status = EG_solidBoolean(source, ebody4, SUBTRACTION, &emodel);
    if (status != EGADS_SUCCESS) goto cleanup;

    status = EG_deleteObject(source);
    if (status != EGADS_SUCCESS) goto cleanup;
    status = EG_deleteObject(ebody4);
    if (status != EGADS_SUCCESS) goto cleanup;

    status = EG_getTopology(emodel, &eref, &oclass, &mtype, data, &nchild, &echilds2,
                &senses);
    if (status != EGADS_SUCCESS) goto cleanup;

    if (oclass != MODEL || nchild != 1) {
        printf("Not a model or are returning more than one body ochild = %d, nchild = %d/n",
            oclass, nchild);
        status = -999;
        goto cleanup;
    }
```

```
            status = EG_copyObject(echilds2[0], NULL, &source);
            if (status != EGADS_SUCCESS) goto cleanup;

            status = EG_deleteObject(emodel);
            if (status != EGADS_SUCCESS) goto cleanup;
        }
        *ebody = source;
    } else {
        *ebody = ebody2;
    }

    /* set the output value(s) */
    status = EG_getMassProperties(*ebody, data);
    if (status != EGADS_SUCCESS) {
        goto cleanup;
    }
    VOLUME(0) = data[0];

    /* remember this model (body) */
#ifdef UDP
    udps[numUdp].ebody = *ebody;
#else
    printf("myVolume = %f\n", myVolume);
#endif

cleanup:
    if (status != EGADS_SUCCESS)
        *string = udpErrorStr(status);

    return status;
}
```