



# Engineering Sketch Pad

## UDP/UDF Programming – Full-featured UDP

### For ESP Rev 1.29

John F. Dannenhoffer, III  
[john@geocentrictech.com](mailto:john@geocentrictech.com)  
Geocentric Technologies LLC

- Use the process from the previous session to create `udpTrain1` from `udpTemplate`
- Modify `train1.c` to:
  - add more arguments
  - revisit top-down build process
  - introduce bottom-up build process
  - add output arguments (OUTPMTRs)
  - compute sensitivities analytically
  - add private data
- Exercise2

# Listing of DOCTEXT within `train1.c` (1)

```

34 #ifdef DOCTEXT
35 UDPRIM train1
36     input Body(s)
37         -NONE-
38     output Body(s)
39         * SolidBody or SheetBody or WireBody
40     input arguments (specified as name/value pairs):
41     -----
42         | argname      | description                                     | default |
43     -----
44         | lenx         | length in x-direction                         | 0       |
45         | leny         | length in y-direction                         | 0       |
46         | lenz         | length in z-direction                         | 0       |
47         | center[3]    | location of the center (or blank)            | 0       |
48     -----
49     output arguments:
50     -----
51         | argname      | description                                     |         |
52     -----
53         | area         | surface area (for SheetBody or SolidBody), otherwise 0 |
54         | volume      | enclosed volume (for SolidBody), otherwise 0         |
55     -----

```

```
56     usage:
57     * serve as a template for UDP developer training
58     * if (all of lenx, leny, lenz are zero)
59         returns an error
60     elseif (two of lenx, leny, lenz are zero)
61         creates a WireBody aligned with a coordinate axis
62     elseif (one of lenx, leny, lenz are zero)
63         creates a SheetBody parallel to a coordinate plane
64     else
65         creates a brick SolidBody
66     * if center[] is not specified, the Body is centered at the origin
67     * analytic sensitivities are computed
68 contributed by:
69     * John Dannenhoffer    john@geocentrictech.com
70 #endif
```

# Listing of train\_1.csm (1)

```

1  # train1_1
2  # written by John Dannenhoffer
3
4  DESPMTR   DX    4.0
5  DESPMTR   DY    2.0
6  DESPMTR   DZ    5.0
7
8  # verify that outputs and their sensitivities are correct
9  OUTPMTR   boxArea
10 OUTPMTR   boxVolume
11 OUTPMTR   xyArea
12 OUTPMTR   xyVolume
13 OUTPMTR   yzArea
14 OUTPMTR   yzVolume
15 OUTPMTR   zxArea
16 OUTPMTR   zxVolume
17
18 # box
19 UDPRIM     train1   lenx DX   leny DY   lenz DZ
20 SET        boxArea  @@area
21 SET        boxVolume @@volume
22
23 # plate parallel to XY plane
24 UDPRIM     train1   lenx DX   leny DY           center 0;0;DZ
25 ATTRIBUTE  _color   $red
26 ATTRIBUTE  _bcolor  $lred
27 SET        xyArea   @@area
28 SET        xyVolume @@volume

```

# Listing of train\_1.csm (2)

```
29 # plate parallel to YZ plane
30 UDPRIM   trainl      leny DY    lenz DZ    center DX;0;0
31 ATTRIBUTE _color      $green
32 ATTRIBUTE _bcolor     $lgreen
33 SET      yzArea      @@area
34 SET      yzVolume    @@volume
35
36 # plate parallel to ZX plane
37 UDPRIM   trainl      lenx DX          lenz DZ    center 0;DY;0
38 ATTRIBUTE _color      $blue
39 ATTRIBUTE _bcolor     $lblue
40 SET      zxArea      @@area
41 SET      zxVolume    @@volume
42
43 # wire in X direction
44 UDPRIM   trainl      lenx DX          center 0;DY;DZ
45
46 # wire in Y direction
47 UDPRIM   trainl          leny DY          center DX;0;DZ
48
49 # wire in Z direction
50 UDPRIM   trainl          lenz DZ          center DX;DY;0
51
52 END
```

- WireBody
  - Node, Curve, Edge, Loop, WireBody
- SheetBody
  - Node, Curve, Edge, Loop, Face, Shell, SheetBody
  - Node, Curve, Edge, Surface, Pcurve, Loop, Face, Shell, SheetBody
- SolidBody
  - Node, Curve, Edge, Loop, Face, Shell, SolidBody
  - Node, Curve, Edge, Surface, Pcurve, Loop, Face, Shell, SolidBody

- Create the Nodes (with `EG_makeTopology`)
  - `egads.pdf` page 89
- Create the Curves (with `EG_makeGeometry`)
  - `egads.pdf` pages 18-29 and 73
- Find the  $t$  values on the Curves associated with the Nodes (with `EG_invEvaluate`)
  - `egads.pdf` page 83
- Create the Edges (with `EG_makeTopology`)
- Create a Loop (with `EG_makeTopology`)
- Create the WireBody (with `EG_makeTopology`)

```
1      int      senses[2];
2      double xyz[10], trange[2], xyzout[3];
3      ego      enodes[3], ecurves[2], eedges[3], eloop, ebody;
4
5      /* make Nodes */
6      xyz[0] = 0;   xyz[1] = 0;   xyz[2] = 0;
7      status = EG_makeTopology(context, NULL, NODE, 0, xyz, 0, NULL,
8                                NULL, &(enodes[0]));
9      CHECK_STATUS(EG_makeTopology);
10
11     xyz[0] = 1;   xyz[1] = 0;   xyz[2] = 0;
12     status = EG_makeTopology(context, NULL, NODE, 0, xyz, 0, NULL,
13                               NULL, &(enodes[1]));
14     CHECK_STATUS(EG_makeTopology);
15
16     xyz[0] = 2;   xyz[1] = 1;   xyz[2] = 0;
17     status = EG_makeTopology(context, NULL, NODE, 0, xyz, 0, NULL,
18                               NULL, &(enodes[2]));
19     CHECK_STATUS(EG_makeTopology);
```

```
1  /* make Curves */
2  xyz[0] = 0;   xyz[1] = 0;   xyz[2] = 0;   /* starting point */
3  xyz[3] = 1;   xyz[4] = 0;   xyz[5] = 0;   /* direction */
4  status = EG_makeGeometry(context, CURVE, LINE, NULL,
5                          NULL, xyz, &(ecurves[0]));
6  CHECK_STATUS(EG_makeGeometry);
7
8  xyz[0] = 1;   xyz[1] = 1;   xyz[2] = 0;   /* center */
9  xyz[3] = 1;   xyz[4] = 0;   xyz[5] = 0;   /* direction 1 */
10 xyz[6] = 0;   xyz[7] = 1;   xyz[8] = 0;   /* direction 2 */
11 xyz[9] = 1;                                     /* radius */
12 status = EG_makeGeometry(context, CURVE, CIRCLE, NULL,
13                          NULL, xyz, &(ecurves[1]));
14 CHECK_STATUS(EG_makeGeometry);
```

```
1  /* find t-range and make Edges */
2  xyz[0] = 0;   xyz[1] = 0;   xyz[2] = 0;
3  status = EG_invEvaluate(ecurve[0], xyz, &(trange[0]), xyzout);
4  CHECK_STATUS(EG_invEvaluate);
5
6  xyz[0] = 1;   xyz[1] = 0;   xyz[2] = 0;
7  status = EG_invEvaluate(ecurve[0], xyz, &(trange[1]), xyzout);
8  CHECK_STATUS(EG_invEvaluate);
9
10 status = EG_makeTopology(context, ecurve[0], EDGE, TWONODE, trange,
11                          2, &(enodes[0]), NULL, &(eedges[0]));
12 CHECK_STATUS(EG_makeTopology);
13
14 xyz[0] = 1;   xyz[1] = 0;   xyz[2] = 0;
15 status = EG_invEvaluate(ecurve[1], xyz, &(trange[0]), xyzout);
16 CHECK_STATUS(EG_invEvaluate);
17
18 xyz[0] = 1;   xyz[1] = 1;   xyz[2] = 0;
19 status = EG_invEvaluate(ecurve[1], xyz, &(trange[1]), xyzout);
20 CHECK_STATUS(EG_invEvaluate);
21
22 status = EG_makeTopology(context, ecurve[1], EDGE, TWONODE, trange,
23                          2, &(enodes[1]), NULL, &(eedges[1]));
24 CHECK_STATUS(EG_makeTopology);
```

```
1  /* make the Loop */
2  senses[0] = SFORWARD;  senses[1] = SFORWARD;
3  status = EG_makeTopology(context, NULL, LOOP, OPEN, NULL,
4                        2, eedges, senses, &eloop);
5  CHECK_STATUS(EG_makeTopology);
6
7  /* make the WireBody */
8  statys = EG_makeTopology(context, NULL, BODY, WIREBODY, NULL,
9                        1, &eloop, NULL, &ebody);
10 CHECK_STATUS(EG_makeTopology);
```

- Create the Nodes (with `EG_makeTopology`)
- Create the Curves (with `EG_makeGeometry`)
- Find the  $t$  values on the Curves associated with the Nodes (with `EG_invEvaluate`)
- Create the Edges (with `EG_makeTopology`)
- Create a Loop (with `EG_makeLoop`)
  - `egads.pdf` page 93
- Create a Face (with `EG_makeFace`)
  - `egads.pdf` page 92
- Create a Shell (with `EG_makeTopology`)
- Create the SheetBody (with `EG_makeTopology`)

- Create the Nodes (with `EG_makeTopology`)
- Create the Curves (with `EG_makeGeometry`)
- Find the  $t$  values on the Curves associated with the Nodes (with `EG_invEvaluate`)
- Create the Edges (with `EG_makeTopology`)
- Create the Surface (with `EG_makeGeometry`)
- Create the Pcurves (with `EG_otherCurve`)
- Create a Loop (with `EG_makeTopology`)
- Create a Face (with `EG_makeTopology`)
- Create a Shell (with `EG_makeTopology`)
- Create the SheetBody (with `EG_makeTopology`)

- Create the Nodes (with `EG_makeTopology`)
- Create the Curves (with `EG_makeGeometry`)
- Find the  $t$  values on the Curves associated with the Nodes (with `EG_invEvaluate`)
- Create the Edges (with `EG_makeTopology`)
- Create the Loops (with `EG_makeLoop`)
- Create the Faces (with `EG_makeFace`)
- Create a Shell (with `EG_makeTopology`)
- Create the SolidBody (with `EG_makeTopology`)

# Listing of `train1.c` (1)

```
1  /*
2  *****
3  *
4  *  udpTrain1 -- training UDP
5  *
6  *          this makes a box, plate, or wire (centered at origin)
7  *          and returns its area and volume (for training purposes)
8  *          sensitivities are computed analytically
9  *
10 *          Written by John Dannenhoffer @ Geocentric Technologies
11 *
12 *****
13 */
14
15 /*
16 * Copyright (C) 2025  John F. Dannenhoffer, III (Geocentric Technologies)
17 *
18 * This library is free software; you can redistribute it and/or
19 * modify it under the terms of the GNU Lesser General Public
20 * License as published by the Free Software Foundation; either
21 * version 2.1 of the License, or (at your option) any later version.
22 *
23 * This library is distributed in the hope that it will be useful,
24 * but WITHOUT ANY WARRANTY; without even the implied warranty of
25 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
26 * Lesser General Public License for more details.
27 *
28 * You should have received a copy of the GNU Lesser General Public
29 * License along with this library; if not, write to the Free Software
30 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
31 * MA 02110-1301 USA
32 */
```

- General notes:
  - the descriptions given focus on the difference between `template.c` (from session04) and `train1.c`
  - at times there will be no explanation slide since the code shown is basically a copy-paste-edit of preceding code
- Lines 4–8: identifying comment (which is different from `template.c`)
- Lines 34–70: contain `DOCTEXT` and were shown above

# Listing of train1.c (2)

```
72  /* uncomment the following to get DEBUG printouts */
73  // #define DEBUG 1
74
75  /* the number of "input" Bodys
76
77     this only needs to be specified if this is a UDF (user-defined
78     function) that consumes Bodys from OpenCSM's stack. (the default
79     value is 0).
80
81     if NUMUDPINPUTBODYS>0 then exactly NUMUDPINPUTBODYS are in emodel
82     if NUMUDPINPUTBODYS<0 then up to -NUMUDPINPUTBODYS are in emodel
83  */
84  #define NUMUDPINPUTBODYS 0
85
86  /* the name of the routine to clean up private data
87
88     this only needs to be specified if the UDP hangs private
89     data onto udps[iudp].data (which is a void*), and that data
90     could not be freed by a simple call to EG_free(). cases where
91     this happens is when udps[iudp].data points to a structure that
92     contains components that had be allocated via separate calls
93     to EG_alloc (or malloc) or which used an allocator other
94     than EG_alloc
95  */
96  #define FREEUDPDATA(A) freePrivateData(A)
97  static int freePrivateData(void *data);
98
99  #define COPYUDPDATA(SRC,TGT) copyPrivateData(SRC,TGT)
100 static int copyPrivateData(/*@null@*/void *src, void **tgt);
```

- Lines 96–100: definitions needed when the UDP/UDF has data that it needs to store for future sensitivity calculations.

# Listing of train1.c (3)

```

101  /* the number of arguments (specified below) */
102  #define NUMUDPARGS 6
103
104  /* set up the necessary structures (uses NUMUDPARGS) */
105  #include "udpUtilities.h"
106
107  /* shorthand macros for accessing argument values and velocities */
108  #define LENX(      IUDP  )  ((double *) (udps[IUDP].arg[0].val)) [0]
109  #define LENX_DOT(  IUDP  )  ((double *) (udps[IUDP].arg[0].dot)) [0]
110  #define LENX_SIZ(  IUDP  )           udps[IUDP].arg[0].size
111
112  #define LENY(      IUDP  )  ((double *) (udps[IUDP].arg[1].val)) [0]
113  #define LENY_DOT(  IUDP  )  ((double *) (udps[IUDP].arg[1].dot)) [0]
114  #define LENY_SIZ(  IUDP  )           udps[IUDP].arg[1].size
115
116  #define LENZ(      IUDP  )  ((double *) (udps[IUDP].arg[2].val)) [0]
117  #define LENZ_DOT(  IUDP  )  ((double *) (udps[IUDP].arg[2].dot)) [0]
118  #define LENZ_SIZ(  IUDP  )           udps[IUDP].arg[2].size
119
120  #define CENTER(    IUDP,I)  ((double *) (udps[IUDP].arg[3].val)) [I]
121  #define CENTER_DOT(IUDP,I)  ((double *) (udps[IUDP].arg[3].dot)) [I]
122  #define CENTER_SIZ(IUDP  )           udps[IUDP].arg[3].size
123
124  #define AREA(      IUDP  )  ((double *) (udps[IUDP].arg[4].val)) [0]
125  #define AREA_DOT(  IUDP  )  ((double *) (udps[IUDP].arg[4].dot)) [0]
126
127  #define VOLUME(    IUDP  )  ((double *) (udps[IUDP].arg[5].val)) [0]
128  #define VOLUME_DOT(IUDP  )  ((double *) (udps[IUDP].arg[5].dot)) [0]

```

- Line 102: there are 4 input and 2 output arguments (for a total of 6)
- Line 109: this defines the velocity (dot) that is need to compute the sensitivities
- Lines 112–128: the macros for the remaining arguments. Note the indicies of the `arg` variable

# Listing of `train1.c` (4)

```

129 /* data about possible arguments
130      argNames: argument name (must be all lower case)
131      argTypes: argument type: +ATTRINT      integer input
132                               -ATTRINT      integer output
133                               +ATTRREAL      double input   (no sensitivities)
134                               -ATTRREAL      double output   (no sensitivities)
135                               +ATTRREALSEN    double input   (with sensitivities)
136                               -ATTRREALSEN    double output   (with sensitivities)
137                               +ATTRSTRING     string input
138                               -ATTRSTRING     *** cannot be used ***
139                               +ATTRFILE       input file
140                               -ATTRFILE       *** cannot be used ***
141                               +ATTRREBUILD    forces rebuild if any variable in
142                                               semi-colon-separated list has changed
143                               -ATTRREBUILD    *** cannot be used ***
144                               +ATTRRECYCLE    forces rebuild by blocking recycling
145                               -ATTRRECYCLE    *** cannot be used ***
146      argIdefs: default value for ATTRINT
147      argDdefs: default value for ATTRREAL or ATTRREALSEN */
148 static char  *argNames[NUMUDPARGS] = {"lenx",      "leny",      "lenz",
149                                       "center",    "area",      "volume",    };
150 static int    argTypes[NUMUDPARGS] = {ATTRREALSEN, ATTRREALSEN, ATTRREALSEN,
151                                       ATTRREALSEN, -ATTRREALSEN, -ATTRREALSEN,};
152 static int    argIdefs[NUMUDPARGS] = {0,          0,          0,
153                                       0,          0,          0,
154                                       0.,         0.,         0.,
155                                       0.,         0.,         0.,    };

```

- Lines 148–149: lists the names of the 6 arguments
- Lines 150–151: `argTypes` set to `ATTRREALSEN` indicate that analytic sensitivities are computed in this UDP/UDF for these input arguments
- Line 151: the `argTypes` for the output arguments are given as `-ATTRREALSEN`, indicating that sensitivities are provided for these outputs by this UDP/UDF

# Listing of `train1.c` (5)

```

156 /* get utility routines: udpErrorStr, udpInitialize, udpReset, udpSet,
157      udpGet, udpVel, udpClean, udpMesh */
158 #include "udpUtilities.c"
159
160 /*
161  *****
162  *
163  *   udpExecute - execute the primitive
164  *
165  *
166  *
167  */
168
169 int
170 udpExecute(ego   context,          /* (in)  EGADS context */
171            ego   *ebody,          /* (out) Body (or model) pointer */
172            int   *nMesh,          /* (out) number of associated meshes */
173            char  *string[])       /* (out) error message */
174 {
175     int      status = EGADS_SUCCESS;
176     int      type=0, sense[4];
177     double   node1[3], node2[3], node3[3], node4[3];
178     double   data[18], trange[2];
179     char     *message=NULL;
180     ego      enodes[4], tempNodes[2], ecurve, eedges[8], eloop, eface;
181     udp_T    *udps = *Udps;
182
183     ROUTINE(udpExecute);
184
185     /* ----- */

```

- Lines 175–177: needed local variables

# Listing of train1.c (6)

```

185 #ifdef DEBUG
186     /* debug printing of the input arguments */
187     printf("udpExecute(context=%llx)\n", (long long)context);
188     printf("lenx(0)      = %f\n", LENX(    0));
189     printf("lenx_dot(0)   = %f\n", LENX_DOT(0));
190     printf("leny(0)      = %f\n", LENY(    0));
191     printf("leny_dot(0)   = %f\n", LENY_DOT(0));
192     printf("lenz(0)      = %f\n", LENZ(    0));
193     printf("lenz_dot(0)   = %f\n", LENZ_DOT(0));
194     if (CENTER_SIZ(0) == 3) {
195         printf("center(0)      = %f %f %f\n", CENTER(    0,0), CENTER(    0,1), CENTER(    0,2));
196         printf("center_dot(0) = %f %f %f\n", CENTER_DOT(0,0), CENTER_DOT(0,1), CENTER_DOT(0,2));
197     }
198 #endif
199
200     /* default return values */
201     *ebody   = NULL;
202     *nMesh   = 0;
203     *string  = NULL;
204
205     /* the place where messages to the user are placed */
206     MALLOC(message, char, 100);
207     message[0] = '\0';
208
209     /* check arguments */
210     if (LENX_SIZ(0) > 1) {
211         snprintf(message, 100, "\"lenx\" should be a scalar");
212         status = EGADS_RANGERR;
213         goto cleanup;

```

- Lines 210–213: the size of the argument is checked to ensure that a scalar (not vector) was provided by ESP during the call

# Listing of `train1.c` (7)

```
214 } else if (LENX(0) < 0) {
215     snprintf(message, 100, "\"lenx\" (=%f) should be non-negative", LENX(0));
216     status = EGADS_RANGERR;
217     goto cleanup;
218
219 } else if (LENY_SIZ(0) > 1) {
220     snprintf(message, 100, "\"leny\" should be a scalar");
221     status = EGADS_RANGERR;
222     goto cleanup;
223
224 } else if (LENY(0) < 0) {
225     snprintf(message, 100, "\"leny\" (=%f) should be non-negative", LENY(0));
226     status = EGADS_RANGERR;
227     goto cleanup;
228
229 } else if (LENZ_SIZ(0) > 1) {
230     snprintf(message, 100, "\"lenz\" should be a scalar");
231     status = EGADS_RANGERR;
232     goto cleanup;
233
234 } else if (LENZ(0) < 0) {
235     snprintf(message, 100, "\"lenz\" (=%f) should be non-negative", LENZ(0));
236     status = EGADS_RANGERR;
237     goto cleanup;
238
239 } else if (LENX(0) <= 0 && LENY(0) <= 0 && LENZ(0) <= 0) {
240     snprintf(message, 100, "cannot have \"lenx\"=\"leny\"=\"lenz\"=0");
241     status = EGADS_GEOMERR;
242     goto cleanup;
243 }
```

# Listing of `train1.c` (8)

```
244     } else if (CENTER_SIZ(0) == 1) {
245         // CENTER is not used
246
247     } else if (CENTER_SIZ(0) != 3) {
248         snprintf(message, 100, "\"center\" should contain 3 entries");
249         status = EGADS_GEOMERR;
250         goto cleanup;
251     }
252
253     /* make private data (not needed here, but included to
254        show how one would do this */
255     if (numUdp == 0) {
256         MALLOC(udps[0].data, char, 30);
257     }
258
259     /* if not after a ocsnCopy (which happens when making finite difference
260        sensitivities), create the string in the private data */
261     if (udps[0].data != NULL) {
262         strcpy((char*)(udps[0].data), "this is test private data");
263     }
264
265     /* cache copy of arguments for future use */
266     status = cacheUdp(NULL);
267     CHECK_STATUS(cacheUdp);
```

- Lines 255–257: if this is the first UDP/UDF of this type (that is `numUdp` is zero), then space is allocated for the private data
- Lines 261–263: during the computation of sensitivities, the entire process is “rebuilt” in order to propagate the velocities. This is done by making a copy of the `Modl`, so in this case the private data should not be copied

# Listing of `train1.c` (9)

```
268 #ifdef DEBUG
269     /* debug printing of cached input arguments */
270     printf("lenx[%d]      = %f\n", numUdp, LENX(      numUdp));
271     printf("lenx_dot[%d]   = %f\n", numUdp, LENX_DOT(numUdp));
272     printf("leny[%d]      = %f\n", numUdp, LENY(      numUdp));
273     printf("leny_dot[%d]   = %f\n", numUdp, LENY_DOT(numUdp));
274     printf("lenz[%d]      = %f\n", numUdp, LENZ(      numUdp));
275     printf("lenz_dot[%d]   = %f\n", numUdp, LENZ_DOT(numUdp));
276     if (CENTER_SIZ(numUdp) == 3) {
277         printf("center[%d]    = %f %f %f\n", numUdp, CENTER(      numUdp,0), CENTER(      numUdp,1),
278         printf("center_dot[%d] = %f %f %f\n", numUdp, CENTER_DOT(numUdp,0), CENTER_DOT(numUdp,1),
279     }
280 #endif
```

# Listing of train1.c (10)

```

281  /* check for 3D SolidBody (and make if requested) */
282  if (LENX(0) > 0 && LENY(0) > 0 && LENZ(0) > 0) {
283
284      /*
285          ^ Y
286          |
287          4----11----8
288          /:          /|
289          3 :          7 |
290          / 4          / 8
291          3----12----7 |
292          | 2-----9|---6 --> X
293          | '          | /
294          2 1          6 5
295          | '          | /
296          1----10----5
297          /
298          Z
299
300      */
301
302      data[0] = -LENX(0)/2;
303      data[1] = -LENY(0)/2;
304      data[2] = -LENZ(0)/2;
305      data[3] = LENX(0);
306      data[4] = LENY(0);
307      data[5] = LENZ(0);

```

- Lines 302–307: set up the data needed by `EG_makeSolidBody` for a box. See `egads.pdf` for a description of the data that is needed for each type of primitive (box, sphere, cylinder, cone, or torus)

```
308      /* move the Body if CENTER has 3 values */
309      if (CENTER_SIZ(0) == 3) {
310          data[0] += CENTER(0,0);
311          data[1] += CENTER(0,1);
312          data[2] += CENTER(0,2);
313      }
314
315      /* make SolidBody */
316      status = EG_makeSolidBody(context, BOX, data, ebody);
317      CHECK_STATUS(EG_makeSolidBody);
318
319      SPLINT_CHECK_FOR_NULL(*ebody);
320
321      /* set the output value(s) */
322      status = EG_getMassProperties(*ebody, data);
323      CHECK_STATUS(EG_getMassProperties);
324
325      AREA( numUdp) = data[1];
326      VOLUME(numUdp) = data[0];
327
328      /* remember this model (Body) */
329      udps[numUdp].ebody = *ebody;
330
331      goto cleanup;
```

- Lines 309–313: if the UDP/UDF is executed without an argument being set, the argument value will be set to a scalar with the default value. So if the `.csm` script does not specify `center`, then `CENTER_SIZ(0)` will be 1. If `center` was set to a 3-vector, the values for the starting-point of the box will need to be modified
- Lines 322–323: the mass properties (including volume and surface area) are computed.
- Lines 325–326: the values of the output arguments (`area` and `volume`) are set to return them to ESP as `@@area` and `@@volume`

# Listing of `train1.c` (12)

```
332  /* WireBody parallel to X axis */
333  } else if (LENY(0) == 0 && LENZ(0) == 0) {
334      type = OCSM_WIRE_BODY;
335      node1[0] = -LENX(0)/2;   node1[1] = 0;           node1[2] = 0;
336      node2[0] = +LENX(0)/2;   node2[1] = 0;           node2[2] = 0;
337
338  /* WireBody parallel to Y axis */
339  } else if (LENZ(0) == 0 && LENX(0) == 0) {
340      type = OCSM_WIRE_BODY;
341      node1[0] = 0;           node1[1] = -LENY(0)/2;   node1[2] = 0;
342      node2[0] = 0;           node2[1] = +LENY(0)/2;   node2[2] = 0;
343
344  /* WireBody parallel to Z axis */
345  } else if (LENX(0) == 0 && LENY(0) == 0) {
346      type = OCSM_WIRE_BODY;
347      node1[0] = 0;           node1[1] = 0;           node1[2] = -LENZ(0)/2;
348      node2[0] = 0;           node2[1] = 0;           node2[2] = +LENZ(0)/2;
```

- Line 334: sets the local variable `type` to indicate that a `WireBody` is to be built
- Lines 335–336: sets the the coordinates of the endpoints of the `WireBody`

```
349 /* SheetBody parallel to XY plane */
350 } else if (LENZ(0) == 0) {
351     type = OCSM_SHEET_BODY;
352     node1[0] = -LENX(0)/2;   node1[1] = -LENY(0)/2;   node1[2] = 0;
353     node2[0] = +LENX(0)/2;   node2[1] = -LENY(0)/2;   node2[2] = 0;
354     node3[0] = +LENX(0)/2;   node3[1] = +LENY(0)/2;   node3[2] = 0;
355     node4[0] = -LENX(0)/2;   node4[1] = +LENY(0)/2;   node4[2] = 0;
356
357 /* SheetBody parallel to YZ plane */
358 } else if (LENX(0) == 0) {
359     type = OCSM_SHEET_BODY;
360     node1[0] = 0;            node1[1] = -LENY(0)/2;   node1[2] = -LENZ(0)/2;
361     node2[0] = 0;            node2[1] = +LENY(0)/2;   node2[2] = -LENZ(0)/2;
362     node3[0] = 0;            node3[1] = +LENY(0)/2;   node3[2] = +LENZ(0)/2;
363     node4[0] = 0;            node4[1] = -LENY(0)/2;   node4[2] = +LENZ(0)/2;
364
365 /* SheetBody parallel to ZX plane */
366 } else if (LENY(0) == 0) {
367     type = OCSM_SHEET_BODY;
368     node1[0] = -LENX(0)/2;   node1[1] = 0;            node1[2] = -LENZ(0)/2;
369     node2[0] = -LENX(0)/2;   node2[1] = 0;            node2[2] = +LENZ(0)/2;
370     node3[0] = +LENX(0)/2;   node3[1] = 0;            node3[2] = +LENZ(0)/2;
371     node4[0] = +LENX(0)/2;   node4[1] = 0;            node4[2] = -LENZ(0)/2;
372 }
```

- Line 351: sets the local variable `type` to indicate that a `SheetBody` is to be built
- Lines 352–355: sets the the coordinates of the corners of the `SheetBody` (which are also the endpoints for the `Edges`)

```
373  /* make the WireBody if required */
374  if (type == OCSM_WIRE_BODY) {
375
376      /* move the Nodes if CENTER has 3 values */
377      if (CENTER_SIZ(0) == 3) {
378          node1[0] += CENTER(0,0);    node1[1] += CENTER(0,1);    node1[2] += CENTER(0,2);
379          node2[0] += CENTER(0,0);    node2[1] += CENTER(0,1);    node2[2] += CENTER(0,2);
380      }
381
382      /* make Nodes */
383      status = EG_makeTopology(context, NULL, NODE, 0, node1, 0, NULL, NULL, &(enodes[0]));
384      CHECK_STATUS(EG_makeTopology);
385
386      status = EG_makeTopology(context, NULL, NODE, 0, node2, 0, NULL, NULL, &(enodes[1]));
387      CHECK_STATUS(EG_makeTopology);
388
389      /* make the Line */
390      data[0] = node1[0];
391      data[1] = node1[1];
392      data[2] = node1[2];
393      data[3] = node2[0] - node1[0];
394      data[4] = node2[1] - node1[1];
395      data[5] = node2[2] - node1[2];
396      status = EG_makeGeometry(context, CURVE, LINE, NULL, NULL, data, &ecurve);
397      CHECK_STATUS(EG_makeGeometry);
398
399      /* get the parameter range */
400      status = EG_invEvaluate(ecurve, node1, &(trange[0]), data);
401      CHECK_STATUS(EG_invEvaluate);
```

- Lines 377–380: adjusts the WireBody endpoints by the values in `center` (if it was set in the `.csm` file)
- Lines 383–387: the Nodes are made given their coordinates
- Lines 390–397: the line Curve is created (see `egads.pdf` for description of the arguments)
- Lines 400–401: the parametric coordinate ( $t$ ) at the Node at the beginning of the Edge that we are going to make is found by inverse evaluation. Note the inverse evaluation is generally slow and somewhat fragile (especially for Surfaces) and should be avoided if possible.

```
402     status = EG_invEvaluate(ecurve, node2, &(trange[1]), data);
403     CHECK_STATUS(EG_invEvaluate);
404
405     /* make Edge */
406     status = EG_makeTopology(context, ecurve, EDGE, TWONODE, trange,
407                             2, enodes, NULL, &(eedges[0]));
408     CHECK_STATUS(EG_makeTopology);
409
410     /* make Loop from this Edge */
411     sense[0] = SFORWARD;
412     status = EG_makeTopology(context, NULL, LOOP, OPEN, NULL, 1, eedges, sense, &eloop);
413     CHECK_STATUS(EG_makeTopology);
414
415     /* create the WireBody (which will be returned) */
416     status = EG_makeTopology(context, NULL, BODY, WIREBODY, NULL, 1, &eloop, NULL, ebody);
417     CHECK_STATUS(EG_makeTopology);
418     if (*ebody == NULL) goto cleanup;    // needed for splint
419
420     /* set the output value(s) */
421     AREA( numUdp) = 0;
422     VOLUME(numUdp) = 0;
423
424     /* remember this model (Body) */
425     udps[numUdp].ebody = *ebody;
```

- Lines 406–408: makes the Edge with the Curve that we made above. In general, Curves have infinite extent (that is there `t` value is unbounded. In contrast, Edges use only a portion of the Curve in the specified `t range`. Also note that the Edge uses a 2-element array of Node `egos`.
- Lines 411–413: a Loop is made from the Edge. Because `sense[0]=SFORWARD`, the direction of the Edge and Loop are the same.
- Lines 416–418: the WireBody is made from the Loop. Arguments that are not needed in any specific call to `EG_makeTopology` are specified as `NULL`.

```

426  /* make the SheetBody if required */
427  } else if (type == OCSM_SHEET_BODY) {
428
429      /*
430          y,z,x
431          ^
432          :
433          4-----<3-----3
434          |       :       |
435          4v     +- - 2^ - -> x,y,z
436          |       |
437          1-----1>-----2
438      */
439
440      /* move the Nodes if CENTER has 3 values */
441      if (CENTER_SIZ(0) == 3) {
442          node1[0] += CENTER(0,0);   node1[1] += CENTER(0,1);   node1[2] += CENTER(0,2);
443          node2[0] += CENTER(0,0);   node2[1] += CENTER(0,1);   node2[2] += CENTER(0,2);
444          node3[0] += CENTER(0,0);   node3[1] += CENTER(0,1);   node3[2] += CENTER(0,2);
445          node4[0] += CENTER(0,0);   node4[1] += CENTER(0,1);   node4[2] += CENTER(0,2);
446      }

```

```
447     /* make Nodes */
448     status = EG_makeTopology(context, NULL, NODE, 0, node1, 0, NULL, NULL, &(enodes[0]));
449     CHECK_STATUS(EG_makeTopology);
450
451     status = EG_makeTopology(context, NULL, NODE, 0, node2, 0, NULL, NULL, &(enodes[1]));
452     CHECK_STATUS(EG_makeTopology);
453
454     status = EG_makeTopology(context, NULL, NODE, 0, node3, 0, NULL, NULL, &(enodes[2]));
455     CHECK_STATUS(EG_makeTopology);
456
457     status = EG_makeTopology(context, NULL, NODE, 0, node4, 0, NULL, NULL, &(enodes[3]));
458     CHECK_STATUS(EG_makeTopology);
```

```
459      /* make the Line 1 */
460      data[0] = node1[0];
461      data[1] = node1[1];
462      data[2] = node1[2];
463      data[3] = node2[0] - node1[0];
464      data[4] = node2[1] - node1[1];
465      data[5] = node2[2] - node1[2];
466      status = EG_makeGeometry(context, CURVE, LINE, NULL, NULL, data, &ecurve);
467      CHECK_STATUS(EG_makeGeometry);
468
469      /* get the parameter range */
470      status = EG_invEvaluate(ecurve, node1, &(trange[0]), data);
471      CHECK_STATUS(EG_invEvaluate);
472
473      status = EG_invEvaluate(ecurve, node2, &(trange[1]), data);
474      CHECK_STATUS(EG_invEvaluate);
475
476      /* make Edge */
477      tempNodes[0] = enodes[0];
478      tempNodes[1] = enodes[1];
479
480      status = EG_makeTopology(context, ecurve, EDGE, TWONODE, trange,
481                               2, tempNodes, NULL, &(eedges[0]));
482      CHECK_STATUS(EG_makeTopology);
```

```
483      /* make the Line 2 */
484      data[0] = node2[0];
485      data[1] = node2[1];
486      data[2] = node2[2];
487      data[3] = node3[0] - node2[0];
488      data[4] = node3[1] - node2[1];
489      data[5] = node3[2] - node2[2];
490      status = EG_makeGeometry(context, CURVE, LINE, NULL, NULL, data, &ecurve);
491      CHECK_STATUS(EG_makeGeometry);
492
493      /* get the parameter range */
494      status = EG_invEvaluate(ecurve, node2, &(trange[0]), data);
495      CHECK_STATUS(EG_invEvaluate);
496
497      status = EG_invEvaluate(ecurve, node3, &(trange[1]), data);
498      CHECK_STATUS(EG_invEvaluate);
499
500      /* make Edge */
501      tempNodes[0] = enodes[1];
502      tempNodes[1] = enodes[2];
503
504      status = EG_makeTopology(context, ecurve, EDGE, TWONODE, trange,
505                               2, tempNodes, NULL, &(eedges[1]));
506      CHECK_STATUS(EG_makeTopology);
```

```
507      /* make the Line 3 */
508      data[0] = node3[0];
509      data[1] = node3[1];
510      data[2] = node3[2];
511      data[3] = node4[0] - node3[0];
512      data[4] = node4[1] - node3[1];
513      data[5] = node4[2] - node3[2];
514      status = EG_makeGeometry(context, CURVE, LINE, NULL, NULL, data, &ecurve);
515      CHECK_STATUS(EG_makeGeometry);
516
517      /* get the parameter range */
518      status = EG_invEvaluate(ecurve, node3, &(trange[0]), data);
519      CHECK_STATUS(EG_invEvaluate);
520
521      status = EG_invEvaluate(ecurve, node4, &(trange[1]), data);
522      CHECK_STATUS(EG_invEvaluate);
523
524      /* make Edge */
525      tempNodes[0] = enodes[2];
526      tempNodes[1] = enodes[3];
527
528      status = EG_makeTopology(context, ecurve, EDGE, TWONODE, trange,
529                               2, tempNodes, NULL, &(eedges[2]));
530      CHECK_STATUS(EG_makeTopology);
```

```
531      /* make the Line 4 */
532      data[0] = node4[0];
533      data[1] = node4[1];
534      data[2] = node4[2];
535      data[3] = node1[0] - node4[0];
536      data[4] = node1[1] - node4[1];
537      data[5] = node1[2] - node4[2];
538      status = EG_makeGeometry(context, CURVE, LINE, NULL, NULL, data, &ecurve);
539      CHECK_STATUS(EG_makeGeometry);
540
541      /* get the parameter range */
542      status = EG_invEvaluate(ecurve, node4, &(trange[0]), data);
543      CHECK_STATUS(EG_invEvaluate);
544
545      status = EG_invEvaluate(ecurve, node1, &(trange[1]), data);
546      CHECK_STATUS(EG_invEvaluate);
547
548      /* make Edge */
549      tempNodes[0] = enodes[3];
550      tempNodes[1] = enodes[0];
551
552      status = EG_makeTopology(context, ecurve, EDGE, TWONODE, trange,
553                               2, tempNodes, NULL, &(eedges[3]));
554      CHECK_STATUS(EG_makeTopology);
```

```
555 /* make Loop from this Edge */
556 sense[0] = SFORWARD;   sense[1] = SFORWARD;
557 sense[2] = SFORWARD;   sense[3] = SFORWARD;
558
559 status = EG_makeTopology(context, NULL, LOOP, CLOSED, NULL, 4, eedges, sense, &eloop)
560 CHECK_STATUS(EG_makeTopology);
561
562 /* make Face from the loop */
563 status = EG_makeFace(eloop, SREVERSE, NULL, &eface);
564 CHECK_STATUS(EG_makeFace);
565
566 /* create the FaceBody (which will be returned) */
567 status = EG_makeTopology(context, NULL, BODY, FACEBODY, NULL, 1, &eface, NULL, &ebody)
568 CHECK_STATUS(EG_makeTopology);
569 if (*ebody == NULL) goto cleanup;    // needed for splint
570
571 /* set the output value(s) */
572 status = EG_getMassProperties(*ebody, data);
573 CHECK_STATUS(EG_getMassProperties);
574
575 AREA( numUdp) = data[1];
576 VOLUME(numUdp) = 0;
577
578 /* remember this model (Body) */
579 udps[numUdp].ebody = *ebody;
```

- Lines 556–560: the Loop is made from the array of Edges created above. Because all the Edges were created with a counter-clockwise sense, they can all have a sense of `SFORWARD`
- Lines 563–564: make a Face given a Loop. The function `EG_makeFace` only applies when the Face is planar. When it is not, one must make a Surface (with `EG_makeGeometry`), make the PCurves (with `EG_otherCurve`), then make the Loop (with `EG_makeTopology`) and then finally the Face (with `EG_makeTopology`)
- Lines 567–569: a `FaceBody` is made from the Face.

```
580     }
581
582 cleanup:
583 #ifdef DEBUG
584     printf("udpExecute -> numUdp=%d, *ebody=%llx\n", numUdp, (long long)(*ebody));
585 #endif
586
587     if (strlen(message) > 0) {
588         *string = message;
589         printf("%s\n", message);
590     } else if (status != EGADS_SUCCESS) {
591         FREE(message);
592         *string = udpErrorStr(status);
593     } else {
594         FREE(message);
595     }
596
597     return status;
598 }
```

# Listing of train1.c (24)

```

599 /*
600 *****
601 *                                                                 *
602 *   udpSensitivity - return sensitivity derivatives for the "real" argument *
603 *                                                                 *
604 * *****
605 */
606
607 int
608 udpSensitivity(ego      ebody,          /* (in)  Body pointer */
609               int      npnt,          /* (in)  number of points */
610               int      entityType,     /* (in)  OCSM entity type */
611               int      entIndex,       /* (in)  OCSM entity index (bias-1) */
612               /*@unused@*/double uvs[], /* (in)  parametric coordinates for evaluation */
613               double vels[])          /* (out) velocities */
614 {
615     int      status = EGADS_SUCCESS;
616
617     int      iudp, judp, i, inode, iedge, iface;
618     double lenx_dot, leny_dot, lenz_dot, xcent_dot, ycent_dot, zcent_dot;
619     double area_dot=0, volume_dot=0;
620
621     ROUTINE(udpSensitivity);
622
623     /* ----- */

```

- Lines 607–613: this defines the function that is called when ESP want sensitivities. It is passed the `ego` of the Body that was created by `udpExecute` and well as a description of the type of sensitivity that is requested.

```
624 #ifdef DEBUG
625     if (uvs != NULL) {
626         printf("udpSensitivity(ebody=%llx, npnt=%d, entType=%d, entIndex=%d, uvs=%f %f)\n",
627             (long long)ebody, npnt, entType, entIndex, uvs[0], uvs[1]);
628     } else {
629         printf("udpSensitivity(ebody=%llx, npnt=%d, entType=%d, entIndex=%d, uvs=NULL)\n",
630             (long long)ebody, npnt, entType, entIndex);
631     }
632 #endif
633
634     /* the following line should be included if sensitivities
635        are not computed analytically */
636     //$$$ return EGADS_NOLOAD;
637
638     /* check that ebody matches one of the ebodys */
639     iudp = 0;
640     for (judp = 1; judp <= numUdp; judp++) {
641         if (ebody == udps[judp].ebody) {
642             iudp = judp;
643             break;
644         }
645     }
646     if (iudp <= 0) {
647         status = EGADS_NOTMODEL;
648         goto cleanup;
649     }
```

- Lines 639–645: find the instance of this UDP. Recall that a UDP can be called several times with (possibly) different arguments. This is the reason we cache the arguments and store the Body ego with the statement `udps[numUdp].ebody=*ebody` at the end of `udpExecute`.
- Lines 646–649: return an error if a match is not found. (Such an error would indicate that something went wrong in ESP and should be reported to the ESP team.)

```
650      /* remember the velocity of center */
651      if (CENTER_SIZ(iudp) == 3) {
652          xcent_dot = CENTER_DOT(iudp,0);
653          ycent_dot = CENTER_DOT(iudp,1);
654          zcent_dot = CENTER_DOT(iudp,2);
655      } else {
656          xcent_dot = 0;
657          ycent_dot = 0;
658          zcent_dot = 0;
659      }
```

- Lines 651–659: find the sensitivity of the center point with respect the sensitivity of the arguments (in this case `CENTER`). Notice that we use `CENTER_DOT(iudp, 0)` because we are finding the sensitivity for the  $iudp^{th}$  instance of this UDP.

# Listing of `train1.c` (27)

```
660  /* WireBody in X direction */
661  if      (LENY(iudp) <= 0 && LENZ(iudp) <= 0) {
662      if (entType == OCSM_NODE) {
663          inode = entIndex;
664
665          if      (inode == 1) {
666              lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = 0;
667          } else if (inode == 2) {
668              lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = 0;
669          } else {
670              status = OCSM_NODE_NOT_FOUND;
671              goto cleanup;
672          }
673      } else if (entType == OCSM_EDGE) {
674          iedge = entIndex;
675
676          if (iedge == 1) {
677              lenx_dot = 0;    leny_dot = 0;    lenz_dot = 0;
678          } else {
679              status = OCSM_EDGE_NOT_FOUND;
680              goto cleanup;
681          }
682      } else {
683          status = OCSM_ILLEGAL_VALUE;
684          goto cleanup;
685      }
```

- Lines 662–672: this call to `udpSensitivity` is asking to compute the sensitivity of a `Node` location. The equations in line 666 and 668 are simply derivatives of the location that was specified in `udpExecute`.
- Lines 673–681: because this UDP creates `WireBodys` that are parallel to a coordinate axis, the sensitivity of any point along the `Edge` (normal to the tangent direction) is the same. So we can just compute a single sensitivity value.

```
686      /* return (constant) velocities, accounting for the movement
687         of the CENTER if it set */
688      for (i = 0; i < npnt; i++) {
689          vels[3*i  ] = lenx_dot + xcent_dot;
690          vels[3*i+1] = leny_dot + ycent_dot;
691          vels[3*i+2] = lenz_dot + zcent_dot;
692      }
```

- Lines 688–692: add the sensitivity of the center point to the sensitivity. This corresponds to lines 377–380.

# Listing of `train1.c` (29)

```
693  /* WireBody in Y direction */
694  } else if (LENZ(iudp) <= 0 && LENX(iudp) <= 0) {
695      if (entType == OCSM_NODE) {
696          inode = entIndex;
697
698          if (inode == 1) {
699              lenx_dot = 0;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = 0;
700          } else if (inode == 2) {
701              lenx_dot = 0;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = 0;
702          } else {
703              status = OCSM_NODE_NOT_FOUND;
704              goto cleanup;
705          }
706      } else if (entType == OCSM_EDGE) {
707          iedge = entIndex;
708
709          if (iedge == 1) {
710              lenx_dot = 0;   leny_dot = 0;   lenz_dot = 0;
711          } else {
712              status = OCSM_EDGE_NOT_FOUND;
713              goto cleanup;
714          }
715      } else {
716          status = OCSM_ILLEGAL_VALUE;
717          goto cleanup;
718      }
```

```
719      /* return (constant) velocities, accounting for the movement
720         of the CENTER if it set */
721      for (i = 0; i < npnt; i++) {
722          vels[3*i  ] = lenx_dot + xcent_dot;
723          vels[3*i+1] = leny_dot + ycent_dot;
724          vels[3*i+2] = lenz_dot + zcent_dot;
725      }
```

# Listing of `train1.c` (31)

```
726  /* WireBody in Z direction */
727  } else if (LENX(iudp) <= 0 && LENY(iudp) <= 0) {
728      if (entType == OCSM_NODE) {
729          inode = entIndex;
730
731          if (inode == 1) {
732              lenx_dot = 0;   leny_dot = 0;   lenz_dot = -LENZ_DOT(iudp)/2;
733          } else if (inode == 2) {
734              lenx_dot = 0;   leny_dot = 0;   lenz_dot = +LENZ_DOT(iudp)/2;
735          } else {
736              status = OCSM_NODE_NOT_FOUND;
737              goto cleanup;
738          }
739      } else if (entType == OCSM_EDGE) {
740          iedge = entIndex;
741
742          if (iedge == 1) {
743              lenx_dot = 0;   leny_dot = 0;   lenz_dot = 0;
744          } else {
745              status = OCSM_EDGE_NOT_FOUND;
746              goto cleanup;
747          }
748      } else {
749          status = OCSM_ILLEGAL_VALUE;
750          goto cleanup;
751      }
```

```
752     /* return (constant) velocities, accounting for the movement
753        of the CENTER if it set */
754     for (i = 0; i < npnt; i++) {
755         vels[3*i] = lenx_dot + xcent_dot;
756         vels[3*i+1] = leny_dot + ycent_dot;
757         vels[3*i+2] = lenz_dot + zcent_dot;
758     }
```

# Listing of train1.c (33)

```

759  /* SheetBody in XY plane (since the velocity on each Node and Edge is
760      a constant, we need to just compute one value) */
761  } else if (LENZ(iudp) <= 0) {
762      if (entType == OCSM_NODE) {
763          inode = entIndex;
764
765          if (inode == 1) {
766              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = 0;
767          } else if (inode == 2) {
768              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = 0;
769          } else if (inode == 3) {
770              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = 0;
771          } else if (inode == 4) {
772              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = 0;
773          } else {
774              status = OCSM_NODE_NOT_FOUND;
775              goto cleanup;
776          }
777      } else if (entType == OCSM_EDGE) {
778          iedge = entIndex;
779
780          if (iedge == 1) {
781              lenx_dot = 0;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = 0;
782          } else if (iedge == 2) {
783              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = 0;   lenz_dot = 0;
784          } else if (iedge == 3) {
785              lenx_dot = 0;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = 0;

```

# Listing of train1.c (34)

```

786         } else if (iedge == 4) {
787             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = 0;
788         } else {
789             status = OCSM_EDGE_NOT_FOUND;
790             goto cleanup;
791         }
792     } else if (entType == OCSM_FACE) {
793         iface = entIndex;
794
795         if (iface == 1) {
796             lenx_dot = 0;    leny_dot = 0;    lenz_dot = 0;
797         } else {
798             status = OCSM_FACE_NOT_FOUND;
799             goto cleanup;
800         }
801     } else {
802         status = OCSM_ILLEGAL_VALUE;
803         goto cleanup;
804     }
805
806     /* return (constant) velocities, accounting for the movement
807        of the CENTER if it set */
808     for (i = 0; i < npnt; i++) {
809         vels[3*i  ] = lenx_dot + xcent_dot;
810         vels[3*i+1] = leny_dot + ycent_dot;
811         vels[3*i+2] = lenz_dot + zcent_dot;
812     }
813
814     /* compute the sensitivities of the area */
815     area_dot = LENX_DOT(iudp) * LENY(iudp) + LENY_DOT(iudp) * LENX(iudp);

```

- Line 815: The area of the SheetBody is given by  $\text{area} = \text{lenx} * \text{leny}$ . Here we analytically differentiate this (using the product rule) to give us  $\text{area\_dot} = \text{lenx\_dot} * \text{leny} + \text{leny\_dot} * \text{lenx}$ . Notice that we use the values and dots for the `iudpth` instance of this UDP.

# Listing of `train1.c` (35)

```
816  /* SheetBody in YZ plane (since the velocity on each Node and Edge is
817      a constant, we need to just compute one value) */
818  } else if (LENX(iudp) <= 0) {
819      if (entType == OCSM_NODE) {
820          inode = entIndex;
821
822          if (inode == 1) {
823              lenx_dot = 0;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = -LENZ_DOT(iudp)/2;
824          } else if (inode == 2) {
825              lenx_dot = 0;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = -LENZ_DOT(iudp)/2;
826          } else if (inode == 3) {
827              lenx_dot = 0;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = +LENZ_DOT(iudp)/2;
828          } else if (inode == 4) {
829              lenx_dot = 0;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = +LENZ_DOT(iudp)/2;
830          } else {
831              status = OCSM_NODE_NOT_FOUND;
832              goto cleanup;
833          }
834      } else if (entType == OCSM_EDGE) {
835          iedge = entIndex;
836
837          if (iedge == 1) {
838              lenx_dot = 0;   leny_dot = 0;   lenz_dot = -LENZ_DOT(iudp)/2;
839          } else if (iedge == 2) {
840              lenx_dot = 0;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = 0;
841          } else if (iedge == 3) {
842              lenx_dot = 0;   leny_dot = 0;   lenz_dot = +LENZ_DOT(iudp)/2;
```

# Listing of train1.c (36)

```

843         } else if (iedge == 4) {
844             lenx_dot = 0;    leny_dot = -LENY_DOT(iudp)/2;    lenz_dot = 0;
845         } else {
846             status = OCSM_EDGE_NOT_FOUND;
847             goto cleanup;
848         }
849     } else if (entType == OCSM_FACE) {
850         iface = entIndex;
851
852         if (iface == 1) {
853             lenx_dot = 0;    leny_dot = 0;    lenz_dot = 0;
854         } else {
855             status = OCSM_FACE_NOT_FOUND;
856             goto cleanup;
857         }
858     } else {
859         status = OCSM_ILLEGAL_VALUE;
860         goto cleanup;
861     }
862
863     /* return (constant) velocities, accounting for the movement
864        of the CENTER if it set */
865     for (i = 0; i < npnt; i++) {
866         vels[3*i  ] = lenx_dot + xcent_dot;
867         vels[3*i+1] = leny_dot + ycent_dot;
868         vels[3*i+2] = lenz_dot + zcent_dot;
869     }
870
871     /* compute the sensitivities of the area */
872     area_dot = LENY_DOT(iudp) * LENZ(iudp) + LENZ_DOT(iudp) * LENY(iudp);

```

# Listing of train1.c (37)

```

873  /* SheetBody in ZX plane (since the velocity on each Node and Edge is
874      a constant, we need to just compute one value) */
875  } else if (LENY(iudp) <= 0) {
876      if (entType == OCSM_NODE) {
877          inode = entIndex;
878
879          if (inode == 1) {
880              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = 0;   lenz_dot = -LENZ_DOT(iudp)/2;
881          } else if (inode == 2) {
882              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = 0;   lenz_dot = +LENZ_DOT(iudp)/2;
883          } else if (inode == 3) {
884              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = 0;   lenz_dot = +LENZ_DOT(iudp)/2;
885          } else if (inode == 4) {
886              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = 0;   lenz_dot = -LENZ_DOT(iudp)/2;
887          } else {
888              status = OCSM_NODE_NOT_FOUND;
889              goto cleanup;
890          }
891      } else if (entType == OCSM_EDGE) {
892          iedge = entIndex;
893
894          if (iedge == 1) {
895              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = 0;   lenz_dot = 0;
896          } else if (iedge == 2) {
897              lenx_dot = 0;   leny_dot = 0;   lenz_dot = +LENZ_DOT(iudp)/2;
898          } else if (iedge == 3) {
899              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = 0;   lenz_dot = 0;

```

# Listing of train1.c (38)

```
900         } else if (iedge == 4) {
901             lenx_dot = 0;    leny_dot = 0;    lenz_dot = -LENZ_DOT(iudp)/2;
902         } else {
903             status = OCSM_EDGE_NOT_FOUND;
904             goto cleanup;
905         }
906     } else if (entType == OCSM_FACE) {
907         iface = entIndex;
908
909         if (iface == 1) {
910             lenx_dot = 0;    leny_dot = 0;    lenz_dot = 0;
911         } else {
912             status = OCSM_FACE_NOT_FOUND;
913             goto cleanup;
914         }
915     }
916 } else {
917     status = OCSM_ILLEGAL_VALUE;
918     goto cleanup;
919 }
920
921 /* return (constant) velocities, accounting for the movement
922    of the CENTER if it set */
923 for (i = 0; i < npnt; i++) {
924     vels[3*i  ] = lenx_dot + xcent_dot;
925     vels[3*i+1] = leny_dot + ycent_dot;
926     vels[3*i+2] = lenz_dot + zcent_dot;
927 }
928
929 /* compute the sensitivities of the area */
930 area_dot = LENZ_DOT(iudp) * LENX_DOT(iudp) + LENX_DOT(iudp) * LENZ_DOT(iudp);
```

# Listing of train1.c (39)

```
931  /* SolidBody (since the velocity on each Node, Edge, and Face is a
932      constant, we need just to compute one value) */
933  } else {
934      if      (entType == OCSM_NODE) {
935          inode = entIndex;
936
937          if      (inode == 1) {
938              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = +L
939          } else if (inode == 2) {
940              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = -L
941          } else if (inode == 3) {
942              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = +L
943          } else if (inode == 4) {
944              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = -L
945          } else if (inode == 5) {
946              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = +L
947          } else if (inode == 6) {
948              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = -L
949          } else if (inode == 7) {
950              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = +L
951          } else if (inode == 8) {
952              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = -L
```

# Listing of train1.c (40)

```

953         } else {
954             status = OCSM_NODE_NOT_FOUND;
955             goto cleanup;
956         }
957     } else if (entType == OCSM_EDGE) {
958         iedge = entIndex;
959
960         if (iedge == 1) {
961             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = -LENY_DOT(iudp)/2;    lenz_dot = 0
962         } else if (iedge == 2) {
963             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = +LENZ_DOT(iudp)/2
964         } else if (iedge == 3) {
965             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = +LENY_DOT(iudp)/2;    lenz_dot = 0
966         } else if (iedge == 4) {
967             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = -LENZ_DOT(iudp)/2
968         } else if (iedge == 5) {
969             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = -LENY_DOT(iudp)/2;    lenz_dot = 0
970         } else if (iedge == 6) {
971             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = +LENZ_DOT(iudp)/2
972         } else if (iedge == 7) {
973             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = +LENY_DOT(iudp)/2;    lenz_dot = 0
974         } else if (iedge == 8) {
975             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = -LENZ_DOT(iudp)/2
976         } else if (iedge == 9) {
977             lenx_dot = 0;    leny_dot = -LENY_DOT(iudp)/2;    lenz_dot = -LENZ_DOT(iudp)/2
978         } else if (iedge == 10) {
979             lenx_dot = 0;    leny_dot = -LENY_DOT(iudp)/2;    lenz_dot = +LENZ_DOT(iudp)/2
980         } else if (iedge == 11) {
981             lenx_dot = 0;    leny_dot = +LENY_DOT(iudp)/2;    lenz_dot = -LENZ_DOT(iudp)/2

```

# Listing of train1.c (41)

```
982         } else if (iedge == 12) {
983             lenx_dot = 0;    leny_dot = +LENY_DOT(iudp)/2;    lenz_dot = +LENZ_DOT(iudp)/2
984         } else {
985             status = OCSM_EDGE_NOT_FOUND;
986             goto cleanup;
987         }
988     } else if (entType == OCSM_FACE) {
989         iface = entIndex;
990
991         if (iface == 1) {
992             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = 0;
993         } else if (iface == 2) {
994             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = 0;
995         } else if (iface == 3) {
996             lenx_dot = 0;    leny_dot = -LENY_DOT(iudp)/2;    lenz_dot = 0;
997         } else if (iface == 4) {
998             lenx_dot = 0;    leny_dot = +LENY_DOT(iudp)/2;    lenz_dot = 0;
999         } else if (iface == 5) {
1000             lenx_dot = 0;    leny_dot = 0;    lenz_dot = -LENZ_DOT(iudp)/2;
1001         } else if (iface == 6) {
1002             lenx_dot = 0;    leny_dot = 0;    lenz_dot = +LENZ_DOT(iudp)/2;
1003         } else {
1004             status = OCSM_FACE_NOT_FOUND;
1005             goto cleanup;
1006         }
1007
1008     } else {
1009         status = OCSM_ILLEGAL_VALUE;
1010         goto cleanup;
1011     }
```

# Listing of train1.c (42)

```
1012     /* return (constant) velocities, accounting for the movement
1013        of the CENTER if it set */
1014     for (i = 0; i < npnt; i++) {
1015         vels[3*i  ] = lenx_dot + xcent_dot;
1016         vels[3*i+1] = leny_dot + ycent_dot;
1017         vels[3*i+2] = lenz_dot + zcent_dot;
1018     }
1019
1020     /* compute the sensitivities of the area and volume */
1021     area_dot = 2 * (LENX_DOT(iudp) * (LENY(iudp) + LENZ(iudp))
1022                   + LENY_DOT(iudp) * (LENZ(iudp) + LENX(iudp))
1023                   + LENZ_DOT(iudp) * (LENX(iudp) + LENY(iudp)));
1024     volume_dot = LENX_DOT(iudp) * LENY(iudp) * LENZ(iudp)
1025                + LENY_DOT(iudp) * LENZ(iudp) * LENX(iudp)
1026                + LENZ_DOT(iudp) * LENX(iudp) * LENY(iudp);
1027 }
1028
1029 AREA_DOT( iudp) = area_dot;
1030 VOLUME_DOT(iudp) = volume_dot;
1031
1032 status = EGADS_SUCCESS;
1033
1034 cleanup:
1035 #ifdef DEBUG
1036     printf("udpSensitivity -> vels=%f %f %f\n", vels[0], vels[1], vels[2]);
1037 #endif
1038     return status;
1039 }
```

```
1040 /*
1041  ****
1042  *
1043  *   freePrivateData - free private data (just an example)
1044  *
1045  ****
1046  */
1047
1048 static int
1049 freePrivateData(void *data)          /* (in)  pointer to private data */
1050 {
1051     int     status = EGADS_SUCCESS;
1052
1053 #ifdef DEBUG
1054     printf("freePrivateData(%s)\n", (char*)(data));
1055 #endif
1056
1057     /* note: this function would not be necessary if we are only calling
1058        EG_free (and then FREEUDPDATA would not be defined above).
1059        It is simply included here to show how one would write
1060        such a function if the allocation was more complicated
1061        than a simple EG_alloc() */
1062
1063     EG_free(data);
1064
1065 //cleanup:
1066     return status;
1067 }
```

- Lines 1048–1067: this routine (and the one that follows) are only needed in the case where the UDP needs to store data. See the comments on lines 1057–1061 for more details.

# Listing of train1.c (44)

```
1068 /*
1069 *****
1070 *
1071 *   copyPrivateData - copy private data (just an example)
1072 *
1073 *****
1074 */
1075
1076 static int
1077 copyPrivateData(
1078     /*@null@*/void *src,          /* (in)  pointer to source private data */
1079     void **tgt)                  /* (in)  pointer to target private data */
1080 {
1081     int    status = EGADS_SUCCESS;
1082
1083     *tgt = NULL;
1084
1085     if (src == NULL) goto cleanup;
1086
1087     /* note: this function would not be necessary if we are only calling
1088        EG_free (and then COPYUDPDATA would not be defined above).
1089        It is simply included here to show how one would write
1090        such a function if the allocation was more complicated
1091        than a simple EG_alloc() */
1092
1093     *tgt = (void *) malloc(sizeof(char)*(strlen((char*)src)+1));
1094     if (*tgt == NULL) {
1095         status = EGADS_MALLOC;
1096         goto cleanup;
1097     }
```

```
1098 #ifdef DEBUG
1099     printf("copyPrivateData(src=%llx, tgt=%llx)\n", (long long)(src), (long long)(*tgt));
1100 #endif
1101
1102     strcpy(*tgt, src);
1103
1104 cleanup:
1105     return status;
1106 }
```

- Make a new UDP (`exercise2`) in the directory (folder) `udpExercise2`
  - the purpose is: create a pyramid whose base is on the  $XY$  plane and which is centered on the  $Z$ -axis. The apex is also on the  $Z$ -axis.
  - the input parameters are:
    - `length` - the length of the base in the  $X$ -direction
    - `width` - the width of the base in the  $Y$ -direction
    - `height` - the  $Z$ -coordinate of the apex
- Make sure that your UDP does appropriate error checking

- First write the UDP so that it uses finite-difference sensitivities
  - draw a picture, numbering the Nodes and Edges
  - make the Nodes (EG\_makeTopology)
  - foreach Edge
    - make a curve (EG\_makeGeometry)
    - find  $t$  at endpoints (EG\_invEvaluate)
    - make the Edge (EG\_makeTopology)
  - foreach Face
    - make a Loop (EG\_makeLoop)
    - make the Face (EG\_makeFace)
  - make a Shell (EG\_makeTopology)
  - make the SolidBody (EG\_makeTopology)

- Notes:

- If you use `EG_makeTopology` to make the Loop instead of `EG_makeLoop`, you need to properly order the Edges and assign the correct sense to each Edge
- If your Face is not planar, instead of `EG_makeFace` you need to first create a Surface (`EG_makeGeometry`), create the PCurves (`EG_otherCurve`), and then the Face (`EG_makeTopology`)
- To aid in debugging, you can use `ocsmPrintEgo(X)` where X is an ego of any type

- Modify the UDP to compute analytic sensitivities
  - For this case

$$\begin{aligned}x &= \text{LENGTH} * f(u, v, w) \\ \dot{x} &= \text{LENGTH\_DOT} * f(u, v, w)\end{aligned}$$

where  $f(u, v, w)$  tells a specific location in the pyramid

- So

$$\dot{x} = x * \frac{\text{LENGTH\_DOT}}{\text{LENGTH}}$$

- The same idea applies in the other coordinate directions

## Area &amp; Volume

$$lh = \sqrt{(l^2/4 + h^2)}$$

$$wh = \sqrt{(w^2/4 + h^2)}$$

$$area = l * w + l * wh + w * lh$$

$$volume = l * w * h / 3$$

## Dots

$$\dot{lh} = (\dot{l} * l / 4 + \dot{h} * h) / lh$$

$$\dot{wh} = (\dot{w} * w / 4 + \dot{h} * h) / wh$$

$$\dot{area} = \dot{l} * w + \dot{w} * l + \dot{w} * lh + w * \dot{lh} + \dot{l} * wh + l * \dot{wh}$$

$$\dot{volume} = (\dot{l} * w * h + \dot{w} * h * l + \dot{h} * l * w) / 3$$