



Engineering Sketch Pad

UDP/UDF Programming – Full-featured UDP

For ESP Rev 1.28

John F. Dannenhoffer, III
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 `train1.hlp` (1)

```
1 UDPRIM train1 -
2   purpose:
3       serve as a template for UDP developer training
4       it creates either:
5           a straight WireBody aligned with a coordinate axis
6           a rectangular SheetBody parallel to a coordinate plane
7           a brick SolidBody
8   input Bodys:
9       -none-
10  input arguments (specified as name/value pairs):
11      lenx      length in x direction          [default 0]
12      leny      length in y direction          [default 0]
13      lenz      length in z direction          [default 0]
14      center    location of the center (3 values) [default 0]
15  output arguments:
16      @@area    surface area (for SheetBody and SolidBody)
17                                     [default 0]
18      @@volume  volume (for SolidBody)         [default 0]
```

```
19  usage notes:
20      lenx, leny, and lenz must be non-negative scalars
21
22      if lenx=leny=lenz=0
23          an error is raised
24      elseif lenx=leny=0
25          a WireBody parallel to the Z-axis is generated
26      elseif leny=lenz=0
27          a WireBody parallel to the X-axis is generated
28      elseif lenz=lenx=0
29          a WireBody parallel to the Y-axis is generated
30      elseif lenx=0
31          a SheetBody parallel to the YZ-plane is generated
32      elseif leny=0
33          a SheetBody parallel to the ZX-plane is generated
34      elseif lenz=0
35          a SheetBody parallel to the XY-plane is generated
36      else
37          a SolidBody is generated
38
39      if center has three values
40          the Body is centered at the specified location
41      else
42          the Body is centered at the origin
43
44      analytic sensitivities are computed
45  contributed by:
46      John Dannenhoffer john@geocentrictech.com
```

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   train1      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   train1      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   train1      lenx DX              center 0;DY;DZ
45
46 # wire in Y direction
47 UDPRIM   train1              leny DY              center DX;0;DZ
48
49 # wire in Z direction
50 UDPRIM   train1              lenz DZ      center DX;DY;0
51
52 END
```

- Create the Nodes (with `EG_makeTopology`)
 - `egads.pdf` page 89
- Create the Curves (with `EG_makeGeometry`)
 - `egads.pdf` pages 18 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 session 1) and `train1.c`
 - at times there will be no explanation slide since the code shown is basically a copy-paste-edit of preceeding code
- Lines 4–8: identifying comment (which is different from `template.c`)

Listing of train1.c (2)

```
33  /* uncomment the following to get DEBUG printouts */
34  //#define DEBUG 1
35
36  /* the number of "input" Bodys
37
38     this only needs to be specified if this is a UDF (user-defined
39     function) that consumes Bodys from OpenCSM's stack. (the default
40     value is 0).
41
42     if NUMUDPINPUTBODYS>0 then exactly NUMUDPINPUTBODYS are in emodel
43     if NUMUDPINPUTBODYS<0 then up to -NUMUDPINPUTBODYS are in emodel
44 */
45 #define NUMUDPINPUTBODYS 0
46
47 /* the name of the routine to clean up private data
48
49     this only needs to be specified if the UDP hangs private
50     data onto udps[iudp].data (which is a void*), and that data
51     could not be freed by a simple call to EG_free(). cases where
52     this happens is when udps[iudp].data points to a structure that
53     contains components that had be allocated via separate calls
54     to EG_alloc (or malloc) or which used an allocator other
55     than EG_alloc
56 */
57 #define FREEUDPDATA(A) freePrivateData(A)
58 static int freePrivateData(void *data);
59
60 #define COPYUDPDATA(SRC,TGT) copyPrivateData(SRC,TGT)
61 static int copyPrivateData(/*@null@*/void *src, void **tgt);
```

- Lines 57–61: definitions needed when the UDP/UDF has data that it needs to store for future sensitivity calculations.

Listing of train1.c (3)

```

62  /* the number of arguments (specified below) */
63  #define NUMUDPARGS 6
64
65  /* set up the necessary structures (uses NUMUDPARGS) */
66  #include "udpUtilities.h"
67
68  /* shorthand macros for accessing argument values and velocities */
69  #define LENX(      IUDP  )  ((double *) (udps[IUDP].arg[0].val)) [0]
70  #define LENX_DOT(  IUDP  )  ((double *) (udps[IUDP].arg[0].dot)) [0]
71  #define LENX_SIZ(  IUDP  )           udps[IUDP].arg[0].size
72
73  #define LENY(      IUDP  )  ((double *) (udps[IUDP].arg[1].val)) [0]
74  #define LENY_DOT(  IUDP  )  ((double *) (udps[IUDP].arg[1].dot)) [0]
75  #define LENY_SIZ(  IUDP  )           udps[IUDP].arg[1].size
76
77  #define LENZ(      IUDP  )  ((double *) (udps[IUDP].arg[2].val)) [0]
78  #define LENZ_DOT(  IUDP  )  ((double *) (udps[IUDP].arg[2].dot)) [0]
79  #define LENZ_SIZ(  IUDP  )           udps[IUDP].arg[2].size
80
81  #define CENTER(    IUDP,I)  ((double *) (udps[IUDP].arg[3].val)) [I]
82  #define CENTER_DOT(IUDP,I)  ((double *) (udps[IUDP].arg[3].dot)) [I]
83  #define CENTER_SIZ(IUDP  )           udps[IUDP].arg[3].size
84
85  #define AREA(      IUDP  )  ((double *) (udps[IUDP].arg[4].val)) [0]
86  #define AREA_DOT(  IUDP  )  ((double *) (udps[IUDP].arg[4].dot)) [0]
87
88  #define VOLUME(    IUDP  )  ((double *) (udps[IUDP].arg[5].val)) [0]
89  #define VOLUME_DOT(IUDP  )  ((double *) (udps[IUDP].arg[5].dot)) [0]

```

- Line 63: there are 4 input and 2 output arguments (for a total of 6)
- Line 70: this defines the velocity (dot) that is need to compute the sensitivities
- Lines 73–89: the macros for the remaining arguments. Note the indicies of the `arg` variable

Listing of `train1.c` (4)

```

90      /* data about possible arguments
91          argNames: argument name (must be all lower case)
92          argTypes: argument type: +ATTRINT       integer input
93                               -ATTRINT        integer output
94                               +ATTRREAL       double input   (no sensitivities)
95                               -ATTRREAL       double output  (no sensitivities)
96                               +ATTRREALSEN    double input   (with sensitivities)
97                               -ATTRREALSEN    double output  (with sensitivities)
98                               +ATTRSTRING     string input
99                               -ATTRSTRING     *** cannot be used ***
100                              +ATTRFILE       input file
101                              -ATTRFILE       *** cannot be used ***
102                              +ATTRREBUILD    forces rebuild if any variable in
103                                              semi-colon-separated list has been changed
104                              -ATTRREBUILD    *** cannot be used ***
105                              +ATTRRECYCLE    forces rebuild (always) by blocking recycling
106                              -ATTRRECYCLE    *** cannot be used ***
107          argIdefs: default value for ATTRINT
108          argDdefs: default value for ATTRREAL or ATTRREALSEN */
109 static char *argNames[NUMUDPARGS] = {"lenx", "leny", "lenz",
110                                     "center", "area", "volume", };
111 static int argTypes[NUMUDPARGS] = {ATTRREALSEN, ATTRREALSEN, ATTRREALSEN,
112                                    ATTRREALSEN, -ATTRREALSEN, -ATTRREALSEN,};
113 static int argIdefs[NUMUDPARGS] = {0, 0, 0,
114                                     0, 0, 0, };
115 static double argDdefs[NUMUDPARGS] = {0., 0., 0.,
116                                         0., 0., 0., };

```

- Lines 109–110: lists the names of the 6 arguments
- Lines 111–112: `argTypes` set to `ATTRREALSEN` indicate that analytic sensitivities are computed in this UDP/UDF for these input arguments
- Line 112: 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)

```

117 /* get utility routines: udpErrorStr, udpInitialize, udpReset, udpSet,
118      udpGet, udpVel, udpClean, udpMesh */
119 #include "udpUtilities.c"
120
121 /*
122  *****
123  *                                                                 *
124  *   udpExecute - execute the primitive                            *
125  *                                                                 *
126  *****
127  */
128
129 int
130 udpExecute(ego   context,          /* (in)  EGADS context */
131            ego   *ebody,          /* (out) Body (or model) pointer */
132            int   *nMesh,          /* (out) number of associated meshes */
133            char  *string[])       /* (out) error message */
134 {
135     int      status = EGADS_SUCCESS;
136     int      type=0, sense[4];
137     double   node1[3], node2[3], node3[3], node4[3];
138     double   data[18], trange[2];
139     char     *message=NULL;
140     ego      enodes[4], tempNodes[2], ecurve, eedges[8], eloop, eface;
141     udp_T    *udps = *Udps;
142
143     ROUTINE(udpExecute);
144
145     /* ----- */

```

- Lines 136–138: needed local variables

Listing of `train1.c` (6)

```

146 #ifdef DEBUG
147     /* debug printing of the input arguments */
148     printf("udpExecute(context=%llx)\n", (long long)context);
149     printf("lenx(0)      = %f\n", LENX(    0));
150     printf("lenx_dot(0)   = %f\n", LENX_DOT(0));
151     printf("leny(0)      = %f\n", LENY(    0));
152     printf("leny_dot(0)   = %f\n", LENY_DOT(0));
153     printf("lenz(0)      = %f\n", LENZ(    0));
154     printf("lenz_dot(0)   = %f\n", LENZ_DOT(0));
155     if (CENTER_SIZ(0) == 3) {
156         printf("center(0)      = %f %f %f\n", CENTER(    0,0), CENTER(    0,1), CENTER(    0,2));
157         printf("center_dot(0) = %f %f %f\n", CENTER_DOT(0,0), CENTER_DOT(0,1), CENTER_DOT(0,2));
158     }
159 #endif
160
161     /* default return values */
162     *ebody = NULL;
163     *nMesh = 0;
164     *string = NULL;
165
166     /* the place where messages to the user are placed */
167     MALLOC(message, char, 100);
168     message[0] = '\0';
169
170     /* check arguments */
171     if (LENX_SIZ(0) > 1) {
172         snprintf(message, 100, "\"lenx\" should be a scalar");
173         status = EGADS_RANGERR;
174         goto cleanup;

```

- Lines 171–174: 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)

```
175 } else if (LENX(0) < 0) {
176     snprintf(message, 100, "\"lenx\" (%f) should be non-negative", LENX(0));
177     status = EGADS_RANGERR;
178     goto cleanup;
179
180 } else if (LENY_SIZ(0) > 1) {
181     snprintf(message, 100, "\"leny\" should be a scalar");
182     status = EGADS_RANGERR;
183     goto cleanup;
184
185 } else if (LENY(0) < 0) {
186     snprintf(message, 100, "\"leny\" (-f) should be non-negative", LENY(0));
187     status = EGADS_RANGERR;
188     goto cleanup;
189
190 } else if (LENZ_SIZ(0) > 1) {
191     snprintf(message, 100, "\"lenz\" should be a scalar");
192     status = EGADS_RANGERR;
193     goto cleanup;
194
195 } else if (LENZ(0) < 0) {
196     snprintf(message, 100, "\"lenz\" (%f) should be non-negative", LENZ(0));
197     status = EGADS_RANGERR;
198     goto cleanup;
199
200 } else if (LENX(0) <= 0 && LENY(0) <= 0 && LENZ(0) <= 0) {
201     snprintf(message, 100, "cannot have \"lenx\"=\"leny\"=\"lenz\"=0");
202     status = EGADS_GEOMERR;
203     goto cleanup;
204
```

Listing of `train1.c` (8)

```
205     } else if (CENTER_SIZ(0) == 1) {
206         // CENTER is not used
207
208     } else if (CENTER_SIZ(0) != 3) {
209         snprintf(message, 100, "\"center\" should contain 3 entries");
210         status = EGADS_GEOMERR;
211         goto cleanup;
212     }
213
214     /* make private data (not needed here, but included to
215        show how one would do this */
216     if (numUdp == 0) {
217         MALLOC(udps[0].data, char, 30);
218     }
219
220     /* if not after a ocsnCopy (which happens when making finite difference
221        sensitivities), create the string in the private data */
222     if (udps[0].data != NULL) {
223         strcpy((char*)(udps[0].data), "this is test private data");
224     }
225
226     /* cache copy of arguments for future use */
227     status = cacheUdp(NULL);
228     CHECK_STATUS(cacheUdp);
```

- Lines 216–218: if this is the first UDP/UDF of this type (that is `numUdp` is zero), then space is allocated for the private data
- Lines 222–224: 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)

```

229 #ifdef DEBUG
230     /* debug printing of cached input arguments */
231     printf("lenx[%d]      = %f\n", numUdp, LENX(      numUdp));
232     printf("lenx_dot[%d]   = %f\n", numUdp, LENX_DOT(numUdp));
233     printf("leny[%d]      = %f\n", numUdp, LENY(      numUdp));
234     printf("leny_dot[%d]   = %f\n", numUdp, LENY_DOT(numUdp));
235     printf("lenz[%d]      = %f\n", numUdp, LENZ(      numUdp));
236     printf("lenz_dot[%d]   = %f\n", numUdp, LENZ_DOT(numUdp));
237     if (CENTER_SIZ(numUdp) == 3) {
238         printf("center[%d]    = %f %f %f\n", numUdp, CENTER(      numUdp,0), CENTER(      numUdp
239         printf("center_dot[%d] = %f %f %f\n", numUdp, CENTER_DOT(numUdp,0), CENTER_DOT(numUdp
240     }
241 #endif

```

Listing of train1.c (10)

```

242  /* check for 3D SolidBody (and make if requested) */
243  if (LENX(0) > 0 && LENY(0) > 0 && LENZ(0) > 0) {
244
245      /*
246          ^ Y
247          |
248          4----11----8
249          /:          /|
250          3 :          7 |
251          / 4          / 8
252          3----12----7 |
253          | 2-----9|---6 --> X
254          | '          | /
255          2 1          6 5
256          |'          | /
257          1----10----5
258          /
259          Z
260
261      */
262
263      data[0] = -LENX(0)/2;
264      data[1] = -LENY(0)/2;
265      data[2] = -LENZ(0)/2;
266      data[3] = LENX(0);
267      data[4] = LENY(0);
268      data[5] = LENZ(0);

```

- Lines 263–268: 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)


```
269      /* move the Body if CENTER has 3 values */
270      if (CENTER_SIZ(0) == 3) {
271          data[0] += CENTER(0,0);
272          data[1] += CENTER(0,1);
273          data[2] += CENTER(0,2);
274      }
275
276      /* make SolidBody */
277      status = EG_makeSolidBody(context, BOX, data, ebody);
278      CHECK_STATUS(EG_makeSolidBody);
279
280      SPLINT_CHECK_FOR_NULL(*ebody);
281
282      /* set the output value(s) */
283      status = EG_getMassProperties(*ebody, data);
284      CHECK_STATUS(EG_getMassProperties);
285
286      AREA( numUdp) = data[1];
287      VOLUME(numUdp) = data[0];
288
289      /* remember this model (Body) */
290      udps[numUdp].ebody = *ebody;
291
292      goto cleanup;
```

- Lines 270–274: 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_SIZE(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 283–284: the mass properties (including volume and surface area) are computed.
- Lines 286–287: 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)

```
293  /* WireBody parallel to X axis */
294  } else if (LENY(0) == 0 && LENZ(0) == 0) {
295      type = OCSM_WIRE_BODY;
296      node1[0] = -LENX(0)/2;   node1[1] = 0;           node1[2] = 0;
297      node2[0] = +LENX(0)/2;   node2[1] = 0;           node2[2] = 0;
298
299  /* WireBody parallel to Y axis */
300  } else if (LENZ(0) == 0 && LENX(0) == 0) {
301      type = OCSM_WIRE_BODY;
302      node1[0] = 0;           node1[1] = -LENY(0)/2;   node1[2] = 0;
303      node2[0] = 0;           node2[1] = +LENY(0)/2;   node2[2] = 0;
304
305  /* WireBody parallel to Z axis */
306  } else if (LENX(0) == 0 && LENY(0) == 0) {
307      type = OCSM_WIRE_BODY;
308      node1[0] = 0;           node1[1] = 0;           node1[2] = -LENZ(0)/2;
309      node2[0] = 0;           node2[1] = 0;           node2[2] = +LENZ(0)/2;
```

- Line 295: sets the local variable `type` to indicate that a `WireBody` is to be built
- Lines 296–297: sets the the coordinates of the endpoints of the `WireBody`

```
310 /* SheetBody parallel to XY plane */
311 } else if (LENZ(0) == 0) {
312     type = OCSM_SHEET_BODY;
313     node1[0] = -LENX(0)/2;   node1[1] = -LENY(0)/2;   node1[2] = 0;
314     node2[0] = +LENX(0)/2;   node2[1] = -LENY(0)/2;   node2[2] = 0;
315     node3[0] = +LENX(0)/2;   node3[1] = +LENY(0)/2;   node3[2] = 0;
316     node4[0] = -LENX(0)/2;   node4[1] = +LENY(0)/2;   node4[2] = 0;
317
318 /* SheetBody parallel to YZ plane */
319 } else if (LENX(0) == 0) {
320     type = OCSM_SHEET_BODY;
321     node1[0] = 0;             node1[1] = -LENY(0)/2;   node1[2] = -LENZ(0)/2;
322     node2[0] = 0;             node2[1] = +LENY(0)/2;   node2[2] = -LENZ(0)/2;
323     node3[0] = 0;             node3[1] = +LENY(0)/2;   node3[2] = +LENZ(0)/2;
324     node4[0] = 0;             node4[1] = -LENY(0)/2;   node4[2] = +LENZ(0)/2;
325
326 /* SheetBody parallel to ZX plane */
327 } else if (LENY(0) == 0) {
328     type = OCSM_SHEET_BODY;
329     node1[0] = -LENX(0)/2;   node1[1] = 0;             node1[2] = -LENZ(0)/2;
330     node2[0] = -LENX(0)/2;   node2[1] = 0;             node2[2] = +LENZ(0)/2;
331     node3[0] = +LENX(0)/2;   node3[1] = 0;             node3[2] = +LENZ(0)/2;
332     node4[0] = +LENX(0)/2;   node4[1] = 0;             node4[2] = -LENZ(0)/2;
333 }
```

- Line 312: sets the local variable `type` to indicate that a `SheetBody` is to be built
- Lines 313–316: sets the the coordinates of the corners of the `SheetBody` (which are also the endpoints for the `Edges`)

```
334  /* make the WireBody if required */
335  if (type == OCSM_WIRE_BODY) {
336
337      /* move the Nodes if CENTER has 3 values */
338      if (CENTER_SIZ(0) == 3) {
339          node1[0] += CENTER(0,0);    node1[1] += CENTER(0,1);    node1[2] += CENTER(0,2);
340          node2[0] += CENTER(0,0);    node2[1] += CENTER(0,1);    node2[2] += CENTER(0,2);
341      }
342
343      /* make Nodes */
344      status = EG_makeTopology(context, NULL, NODE, 0, node1, 0, NULL, NULL, &(enodes[0]));
345      CHECK_STATUS(EG_makeTopology);
346
347      status = EG_makeTopology(context, NULL, NODE, 0, node2, 0, NULL, NULL, &(enodes[1]));
348      CHECK_STATUS(EG_makeTopology);
349
350      /* make the Line */
351      data[0] = node1[0];
352      data[1] = node1[1];
353      data[2] = node1[2];
354      data[3] = node2[0] - node1[0];
355      data[4] = node2[1] - node1[1];
356      data[5] = node2[2] - node1[2];
357      status = EG_makeGeometry(context, CURVE, LINE, NULL, NULL, data, &ecurve);
358      CHECK_STATUS(EG_makeGeometry);
359
360      /* get the parameter range */
361      status = EG_invEvaluate(ecurve, node1, &(trange[0]), data);
362      CHECK_STATUS(EG_invEvaluate);
```

- Lines 338–341: adjusts the WireBody endpoints by the values in `center` (if it was set in the `.csm` file)
- Lines 344–348: the Nodes are made given their coordinates
- Lines 351–358: the line Curve is created (see `egads.pdf` for description of the arguments)
- Lines 361–362: 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.


```
363     status = EG_invEvaluate(ecurve, node2, &(trange[1]), data);
364     CHECK_STATUS(EG_invEvaluate);
365
366     /* make Edge */
367     status = EG_makeTopology(context, ecurve, EDGE, TWONODE, trange,
368                             2, enodes, NULL, &(eedges[0]));
369     CHECK_STATUS(EG_makeTopology);
370
371     /* make Loop from this Edge */
372     sense[0] = SFORWARD;
373     status = EG_makeTopology(context, NULL, LOOP, OPEN, NULL, 1, eedges, sense, &eloop);
374     CHECK_STATUS(EG_makeTopology);
375
376     /* create the WireBody (which will be returned) */
377     status = EG_makeTopology(context, NULL, BODY, WIREBODY, NULL, 1, &eloop, NULL, ebody);
378     CHECK_STATUS(EG_makeTopology);
379     if (*ebody == NULL) goto cleanup;    // needed for splint
380
381     /* set the output value(s) */
382     AREA( numUdp) = 0;
383     VOLUME(numUdp) = 0;
384
385     /* remember this model (Body) */
386     udps[numUdp].ebody = *ebody;
```

- Lines 367–369: 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 372–374: a Loop is made from the Edge. Because `sense[0]=SFORWARD`, the direction of the Edge and Loop are the same.
- Lines 377–379: the WireBody is made from the Loop. Arguments that are not needed in any specific call to `EG_makeTopology` are specified as `NULL`.

Listing of train1.c (16)

```

387  /* make the SheetBody if required */
388  } else if (type == OCSM_SHEET_BODY) {
389
390      /*
391          y,z,x
392          ^
393          :
394          4-----<3-----3
395          |       :       |
396          4v     +- - 2^ - -> x,y,z
397          |       |
398          1-----1>-----2
399      */
400
401      /* move the Nodes if CENTER has 3 values */
402      if (CENTER_SIZ(0) == 3) {
403          node1[0] += CENTER(0,0);    node1[1] += CENTER(0,1);    node1[2] += CENTER(0,2);
404          node2[0] += CENTER(0,0);    node2[1] += CENTER(0,1);    node2[2] += CENTER(0,2);
405          node3[0] += CENTER(0,0);    node3[1] += CENTER(0,1);    node3[2] += CENTER(0,2);
406          node4[0] += CENTER(0,0);    node4[1] += CENTER(0,1);    node4[2] += CENTER(0,2);
407      }

```

```
408     /* make Nodes */
409     status = EG_makeTopology(context, NULL, NODE, 0, node1, 0, NULL, NULL, &(enodes[0]));
410     CHECK_STATUS(EG_makeTopology);
411
412     status = EG_makeTopology(context, NULL, NODE, 0, node2, 0, NULL, NULL, &(enodes[1]));
413     CHECK_STATUS(EG_makeTopology);
414
415     status = EG_makeTopology(context, NULL, NODE, 0, node3, 0, NULL, NULL, &(enodes[2]));
416     CHECK_STATUS(EG_makeTopology);
417
418     status = EG_makeTopology(context, NULL, NODE, 0, node4, 0, NULL, NULL, &(enodes[3]));
419     CHECK_STATUS(EG_makeTopology);
```

Listing of `train1.c` (18)

```

420      /* make the Line 1 */
421      data[0] = node1[0];
422      data[1] = node1[1];
423      data[2] = node1[2];
424      data[3] = node2[0] - node1[0];
425      data[4] = node2[1] - node1[1];
426      data[5] = node2[2] - node1[2];
427      status = EG_makeGeometry(context, CURVE, LINE, NULL, NULL, data, &ecurve);
428      CHECK_STATUS(EG_makeGeometry);
429
430      /* get the parameter range */
431      status = EG_invEvaluate(ecurve, node1, &(trange[0]), data);
432      CHECK_STATUS(EG_invEvaluate);
433
434      status = EG_invEvaluate(ecurve, node2, &(trange[1]), data);
435      CHECK_STATUS(EG_invEvaluate);
436
437      /* make Edge */
438      tempNodes[0] = enodes[0];
439      tempNodes[1] = enodes[1];
440
441      status = EG_makeTopology(context, ecurve, EDGE, TWONODE, trange,
442                               2, tempNodes, NULL, &(eedges[0]));
443      CHECK_STATUS(EG_makeTopology);

```

Listing of `train1.c` (19)

```
444      /* make the Line 2 */
445      data[0] = node2[0];
446      data[1] = node2[1];
447      data[2] = node2[2];
448      data[3] = node3[0] - node2[0];
449      data[4] = node3[1] - node2[1];
450      data[5] = node3[2] - node2[2];
451      status = EG_makeGeometry(context, CURVE, LINE, NULL, NULL, data, &ecurve);
452      CHECK_STATUS(EG_makeGeometry);
453
454      /* get the parameter range */
455      status = EG_invEvaluate(ecurve, node2, &(trange[0]), data);
456      CHECK_STATUS(EG_invEvaluate);
457
458      status = EG_invEvaluate(ecurve, node3, &(trange[1]), data);
459      CHECK_STATUS(EG_invEvaluate);
460
461      /* make Edge */
462      tempNodes[0] = enodes[1];
463      tempNodes[1] = enodes[2];
464
465      status = EG_makeTopology(context, ecurve, EDGE, TWONODE, trange,
466                               2, tempNodes, NULL, &(eedges[1]));
467      CHECK_STATUS(EG_makeTopology);
```

```
468      /* make the Line 3 */
469      data[0] = node3[0];
470      data[1] = node3[1];
471      data[2] = node3[2];
472      data[3] = node4[0] - node3[0];
473      data[4] = node4[1] - node3[1];
474      data[5] = node4[2] - node3[2];
475      status = EG_makeGeometry(context, CURVE, LINE, NULL, NULL, data, &ecurve);
476      CHECK_STATUS(EG_makeGeometry);
477
478      /* get the parameter range */
479      status = EG_invEvaluate(ecurve, node3, &(trange[0]), data);
480      CHECK_STATUS(EG_invEvaluate);
481
482      status = EG_invEvaluate(ecurve, node4, &(trange[1]), data);
483      CHECK_STATUS(EG_invEvaluate);
484
485      /* make Edge */
486      tempNodes[0] = enodes[2];
487      tempNodes[1] = enodes[3];
488
489      status = EG_makeTopology(context, ecurve, EDGE, TWONODE, trange,
490                               2, tempNodes, NULL, &(eedges[2]));
491      CHECK_STATUS(EG_makeTopology);
```

```
492      /* make the Line 4 */
493      data[0] = node4[0];
494      data[1] = node4[1];
495      data[2] = node4[2];
496      data[3] = node1[0] - node4[0];
497      data[4] = node1[1] - node4[1];
498      data[5] = node1[2] - node4[2];
499      status = EG_makeGeometry(context, CURVE, LINE, NULL, NULL, data, &ecurve);
500      CHECK_STATUS(EG_makeGeometry);
501
502      /* get the parameter range */
503      status = EG_invEvaluate(ecurve, node4, &(trange[0]), data);
504      CHECK_STATUS(EG_invEvaluate);
505
506      status = EG_invEvaluate(ecurve, node1, &(trange[1]), data);
507      CHECK_STATUS(EG_invEvaluate);
508
509      /* make Edge */
510      tempNodes[0] = enodes[3];
511      tempNodes[1] = enodes[0];
512
513      status = EG_makeTopology(context, ecurve, EDGE, TWONODE, trange,
514                               2, tempNodes, NULL, &(eedges[3]));
515      CHECK_STATUS(EG_makeTopology);
```



```
516      /* make Loop from this Edge */
517      sense[0] = SFORWARD;   sense[1] = SFORWARD;
518      sense[2] = SFORWARD;   sense[3] = SFORWARD;
519
520      status = EG_makeTopology(context, NULL, LOOP, CLOSED, NULL, 4, eedges, sense, &eloop)
521      CHECK_STATUS(EG_makeTopology);
522
523      /* make Face from the loop */
524      status = EG_makeFace(eloop, SREVERSE, NULL, &eface);
525      CHECK_STATUS(EG_makeFace);
526
527      /* create the FaceBody (which will be returned) */
528      status = EG_makeTopology(context, NULL, BODY, FACEBODY, NULL, 1, &eface, NULL, &ebody)
529      CHECK_STATUS(EG_makeTopology);
530      if (*ebody == NULL) goto cleanup;    // needed for splint
531
532      /* set the output value(s) */
533      status = EG_getMassProperties(*ebody, data);
534      CHECK_STATUS(EG_getMassProperties);
535
536      AREA( numUdp) = data[1];
537      VOLUME(numUdp) = 0;
538
539      /* remember this model (Body) */
540      udps[numUdp].ebody = *ebody;
```

- Lines 517–521: 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 524–525: 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 528–530: a `FaceBody` is made from the Face.

```
541     }
542
543 cleanup:
544 #ifdef DEBUG
545     printf("udpExecute -> numUdp=%d, *ebody=%llx\n", numUdp, (long long)(*ebody));
546 #endif
547
548     if (strlen(message) > 0) {
549         *string = message;
550         printf("%s\n", message);
551     } else if (status != EGADS_SUCCESS) {
552         FREE(message);
553         *string = udpErrorStr(status);
554     } else {
555         FREE(message);
556     }
557
558     return status;
559 }
```

Listing of train1.c (24)

```

560 /*
561 *****
562 *
563 *   udpSensitivity - return sensitivity derivatives for the "real" argument *
564 *
565 *****
566 */
567
568 int
569 udpSensitivity(ego      ebody,          /* (in)  Body pointer */
570               int      npnt,          /* (in)  number of points */
571               int      entType,        /* (in)  OCSM entity type */
572               int      entIndex,       /* (in)  OCSM entity index (bias-1) */
573               /*@unused@*/double uvs[], /* (in)  parametric coordinates for evaluation */
574               double vels[])           /* (out) velocities */
575 {
576     int      status = EGADS_SUCCESS;
577
578     int      iudp, judp, i, inode, iedge, iface;
579     double lenx_dot, leny_dot, lenz_dot, xcent_dot, ycent_dot, zcent_dot;
580     double area_dot=0, volume_dot=0;
581
582     ROUTINE(udpSensitivity);
583
584     /* ----- */

```

- Lines 568–574: 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.

```
585 #ifdef DEBUG
586     if (uvs != NULL) {
587         printf("udpSensitivity(ebody=%llx, npnt=%d, entType=%d, entIndex=%d, uvs=%f %f)\n",
588             (long long)ebody, npnt, entType, entIndex, uvs[0], uvs[1]);
589     } else {
590         printf("udpSensitivity(ebody=%llx, npnt=%d, entType=%d, entIndex=%d, uvs=NULL)\n",
591             (long long)ebody, npnt, entType, entIndex);
592     }
593 #endif
594
595     /* the following line should be included if sensitivities
596        are not computed analytically */
597     //$$$    return EGADS_NOLOAD;
598
599     /* check that ebody matches one of the ebodys */
600     iudp = 0;
601     for (judp = 1; judp <= numUdp; judp++) {
602         if (ebody == udps[judp].ebody) {
603             iudp = judp;
604             break;
605         }
606     }
607     if (iudp <= 0) {
608         status = EGADS_NOTMODEL;
609         goto cleanup;
610     }
```

- Lines 600–606: 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 607–610: 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.)

```
611  /* remember the velocity of center */
612  if (CENTER_SIZ(iudp) == 3) {
613      xcent_dot = CENTER_DOT(iudp,0);
614      ycent_dot = CENTER_DOT(iudp,1);
615      zcent_dot = CENTER_DOT(iudp,2);
616  } else {
617      xcent_dot = 0;
618      ycent_dot = 0;
619      zcent_dot = 0;
620  }
```


- Lines 612–620: 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 `iudpth` instance of this UDP.

```
621  /* WireBody in X direction */
622  if      (LENY(iudp) <= 0 && LENZ(iudp) <= 0) {
623      if (entType == OCSM_NODE) {
624          inode = entIndex;
625
626          if      (inode == 1) {
627              lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = 0;
628          } else if (inode == 2) {
629              lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = 0;
630          } else {
631              status = OCSM_NODE_NOT_FOUND;
632              goto cleanup;
633          }
634      } else if (entType == OCSM_EDGE) {
635          iedge = entIndex;
636
637          if (iedge == 1) {
638              lenx_dot = 0;    leny_dot = 0;    lenz_dot = 0;
639          } else {
640              status = OCSM_EDGE_NOT_FOUND;
641              goto cleanup;
642          }
643      } else {
644          status = OCSM_ILLEGAL_VALUE;
645          goto cleanup;
646      }
```

- Lines 623–633: this call to `udpSensitivity` is asking to compute the sensitivity of a `Node` location. The equations in line 627 and 629 are simply derivatives of the location that was specified in `udpExecute`.
- Lines 634–642: 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.

```
647     /* return (constant) velocities, accounting for the movement
648        of the CENTER if it set */
649     for (i = 0; i < npnt; i++) {
650         vels[3*i  ] = lenx_dot + xcent_dot;
651         vels[3*i+1] = leny_dot + ycent_dot;
652         vels[3*i+2] = lenz_dot + zcent_dot;
653     }
```

- Lines 649–653: add the sensitivity of the center point to the sensitivity. This corresponds to lines 338–341.

```
654  /* WireBody in Y direction */
655  } else if (LENZ(iudp) <= 0 && LENX(iudp) <= 0) {
656      if (entType == OCSM_NODE) {
657          inode = entIndex;
658
659          if (inode == 1) {
660              lenx_dot = 0;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = 0;
661          } else if (inode == 2) {
662              lenx_dot = 0;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = 0;
663          } else {
664              status = OCSM_NODE_NOT_FOUND;
665              goto cleanup;
666          }
667      } else if (entType == OCSM_EDGE) {
668          iedge = entIndex;
669
670          if (iedge == 1) {
671              lenx_dot = 0;   leny_dot = 0;   lenz_dot = 0;
672          } else {
673              status = OCSM_EDGE_NOT_FOUND;
674              goto cleanup;
675          }
676      } else {
677          status = OCSM_ILLEGAL_VALUE;
678          goto cleanup;
679      }
```

```
680      /* return (constant) velocities, accounting for the movement
681         of the CENTER if it set */
682      for (i = 0; i < npnt; i++) {
683          vels[3*i  ] = lenx_dot + xcent_dot;
684          vels[3*i+1] = leny_dot + ycent_dot;
685          vels[3*i+2] = lenz_dot + zcent_dot;
686      }
```

Listing of `train1.c` (31)

```
687  /* WireBody in Z direction */
688  } else if (LENX(iudp) <= 0 && LENY(iudp) <= 0) {
689      if (entType == OCSM_NODE) {
690          inode = entIndex;
691
692          if (inode == 1) {
693              lenx_dot = 0;   leny_dot = 0;   lenz_dot = -LENZ_DOT(iudp)/2;
694          } else if (inode == 2) {
695              lenx_dot = 0;   leny_dot = 0;   lenz_dot = +LENZ_DOT(iudp)/2;
696          } else {
697              status = OCSM_NODE_NOT_FOUND;
698              goto cleanup;
699          }
700      } else if (entType == OCSM_EDGE) {
701          iedge = entIndex;
702
703          if (iedge == 1) {
704              lenx_dot = 0;   leny_dot = 0;   lenz_dot = 0;
705          } else {
706              status = OCSM_EDGE_NOT_FOUND;
707              goto cleanup;
708          }
709      } else {
710          status = OCSM_ILLEGAL_VALUE;
711          goto cleanup;
712      }
```



```
713     /* return (constant) velocities, accounting for the movement
714        of the CENTER if it set */
715     for (i = 0; i < npnt; i++) {
716         vels[3*i  ] = lenx_dot + xcent_dot;
717         vels[3*i+1] = leny_dot + ycent_dot;
718         vels[3*i+2] = lenz_dot + zcent_dot;
719     }
```

Listing of `train1.c` (33)

```
720  /* SheetBody in XY plane (since the velocity on each Node and Edge is
721     a constant, we need to just compute one value) */
722  } else if (LENZ(iudp) <= 0) {
723      if (entType == OCSM_NODE) {
724          inode = entIndex;
725
726          if (inode == 1) {
727              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = 0;
728          } else if (inode == 2) {
729              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = 0;
730          } else if (inode == 3) {
731              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = 0;
732          } else if (inode == 4) {
733              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = 0;
734          } else {
735              status = OCSM_NODE_NOT_FOUND;
736              goto cleanup;
737          }
738      } else if (entType == OCSM_EDGE) {
739          iedge = entIndex;
740
741          if (iedge == 1) {
742              lenx_dot = 0;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = 0;
743          } else if (iedge == 2) {
744              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = 0;   lenz_dot = 0;
745          } else if (iedge == 3) {
746              lenx_dot = 0;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = 0;
```

Listing of train1.c (34)

```

747         } else if (iedge == 4) {
748             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = 0;
749         } else {
750             status = OCSM_EDGE_NOT_FOUND;
751             goto cleanup;
752         }
753     } else if (entType == OCSM_FACE) {
754         iface = entIndex;
755
756         if (iface == 1) {
757             lenx_dot = 0;    leny_dot = 0;    lenz_dot = 0;
758         } else {
759             status = OCSM_FACE_NOT_FOUND;
760             goto cleanup;
761         }
762     } else {
763         status = OCSM_ILLEGAL_VALUE;
764         goto cleanup;
765     }
766
767     /* return (constant) velocities, accounting for the movement
768        of the CENTER if it set */
769     for (i = 0; i < npnt; i++) {
770         vels[3*i  ] = lenx_dot + xcent_dot;
771         vels[3*i+1] = leny_dot + ycent_dot;
772         vels[3*i+2] = lenz_dot + zcent_dot;
773     }
774
775     /* compute the sensitivities of the area */
776     area_dot = LENX_DOT(iudp) * LENY(iudp) + LENY_DOT(iudp) * LENX(iudp);

```

- Line 776: 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)

```

777  /* SheetBody in YZ plane (since the velocity on each Node and Edge is
778     a constant, we need to just compute one value) */
779  } else if (LENX(iudp) <= 0) {
780      if (entType == OCSM_NODE) {
781          inode = entIndex;
782
783          if (inode == 1) {
784              lenx_dot = 0;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = -LENZ_DOT(iudp)/2;
785          } else if (inode == 2) {
786              lenx_dot = 0;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = -LENZ_DOT(iudp)/2;
787          } else if (inode == 3) {
788              lenx_dot = 0;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = +LENZ_DOT(iudp)/2;
789          } else if (inode == 4) {
790              lenx_dot = 0;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = +LENZ_DOT(iudp)/2;
791          } else {
792              status = OCSM_NODE_NOT_FOUND;
793              goto cleanup;
794          }
795      } else if (entType == OCSM_EDGE) {
796          iedge = entIndex;
797
798          if (iedge == 1) {
799              lenx_dot = 0;   leny_dot = 0;   lenz_dot = -LENZ_DOT(iudp)/2;
800          } else if (iedge == 2) {
801              lenx_dot = 0;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = 0;
802          } else if (iedge == 3) {
803              lenx_dot = 0;   leny_dot = 0;   lenz_dot = +LENZ_DOT(iudp)/2;

```

Listing of train1.c (36)

```
804         } else if (iedge == 4) {
805             lenx_dot = 0;    leny_dot = -LENY_DOT(iudp)/2;    lenz_dot = 0;
806         } else {
807             status = OCSM_EDGE_NOT_FOUND;
808             goto cleanup;
809         }
810     } else if (entType == OCSM_FACE) {
811         iface = entIndex;
812
813         if (iface == 1) {
814             lenx_dot = 0;    leny_dot = 0;    lenz_dot = 0;
815         } else {
816             status = OCSM_FACE_NOT_FOUND;
817             goto cleanup;
818         }
819     } else {
820         status = OCSM_ILLEGAL_VALUE;
821         goto cleanup;
822     }
823
824     /* return (constant) velocities, accounting for the movement
825        of the CENTER if it set */
826     for (i = 0; i < npnt; i++) {
827         vels[3*i ] = lenx_dot + xcent_dot;
828         vels[3*i+1] = leny_dot + ycent_dot;
829         vels[3*i+2] = lenz_dot + zcent_dot;
830     }
831
832     /* compute the sensitivities of the area */
833     area_dot = LENY_DOT(iudp) * LENZ(iudp) + LENZ_DOT(iudp) * LENY(iudp);
```

Listing of train1.c (37)

```
834 /* SheetBody in ZX plane (since the velocity on each Node and Edge is
835      a constant, we need to just compute one value) */
836 } else if (LENY(iudp) <= 0) {
837     if (entType == OCSM_NODE) {
838         inode = entIndex;
839
840         if (inode == 1) {
841             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = -LENZ_DOT(iudp)/2;
842         } else if (inode == 2) {
843             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = +LENZ_DOT(iudp)/2;
844         } else if (inode == 3) {
845             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = +LENZ_DOT(iudp)/2;
846         } else if (inode == 4) {
847             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = -LENZ_DOT(iudp)/2;
848         } else {
849             status = OCSM_NODE_NOT_FOUND;
850             goto cleanup;
851         }
852     } else if (entType == OCSM_EDGE) {
853         iedge = entIndex;
854
855         if (iedge == 1) {
856             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = 0;
857         } else if (iedge == 2) {
858             lenx_dot = 0;    leny_dot = 0;    lenz_dot = +LENZ_DOT(iudp)/2;
859         } else if (iedge == 3) {
860             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = 0;
```

Listing of train1.c (38)

```
861         } else if (iedge == 4) {
862             lenx_dot = 0;    leny_dot = 0;    lenz_dot = -LENZ_DOT(iudp)/2;
863         } else {
864             status = OCSM_EDGE_NOT_FOUND;
865             goto cleanup;
866         }
867     } else if (entType == OCSM_FACE) {
868         iface = entIndex;
869
870         if (iface == 1) {
871             lenx_dot = 0;    leny_dot = 0;    lenz_dot = 0;
872         } else {
873             status = OCSM_FACE_NOT_FOUND;
874             goto cleanup;
875         }
876
877     } else {
878         status = OCSM_ILLEGAL_VALUE;
879         goto cleanup;
880     }
881
882     /* return (constant) velocities, accounting for the movement
883        of the CENTER if it set */
884     for (i = 0; i < npnt; i++) {
885         vels[3*i  ] = lenx_dot + xcent_dot;
886         vels[3*i+1] = leny_dot + ycent_dot;
887         vels[3*i+2] = lenz_dot + zcent_dot;
888     }
889
890     /* compute the sensitivities of the area */
891     area_dot = LENZ_DOT(iudp) * LENX_DOT(iudp) + LENX_DOT(iudp) * LENZ_DOT(iudp);
```


Listing of `train1.c` (39)

```
892  /* SolidBody (since the velocity on each Node, Edge, and Face is a
893      constant, we need just to compute one value) */
894  } else {
895      if      (entType == OCSM_NODE) {
896          inode = entIndex;
897
898          if      (inode == 1) {
899              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = +L
900          } else if (inode == 2) {
901              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = -L
902          } else if (inode == 3) {
903              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = +L
904          } else if (inode == 4) {
905              lenx_dot = -LENX_DOT(iudp)/2;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = -L
906          } else if (inode == 5) {
907              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = +L
908          } else if (inode == 6) {
909              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = -LENY_DOT(iudp)/2;   lenz_dot = -L
910          } else if (inode == 7) {
911              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = +L
912          } else if (inode == 8) {
913              lenx_dot = +LENX_DOT(iudp)/2;   leny_dot = +LENY_DOT(iudp)/2;   lenz_dot = -L
```

Listing of train1.c (40)

```

914         } else {
915             status = OCSM_NODE_NOT_FOUND;
916             goto cleanup;
917         }
918     } else if (entType == OCSM_EDGE) {
919         iedge = entIndex;
920
921         if (iedge == 1) {
922             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = -LENY_DOT(iudp)/2;    lenz_dot = 0
923         } else if (iedge == 2) {
924             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = +LENZ_DOT(iudp)/2
925         } else if (iedge == 3) {
926             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = +LENY_DOT(iudp)/2;    lenz_dot = 0
927         } else if (iedge == 4) {
928             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = -LENZ_DOT(iudp)/2
929         } else if (iedge == 5) {
930             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = -LENY_DOT(iudp)/2;    lenz_dot = 0
931         } else if (iedge == 6) {
932             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = +LENZ_DOT(iudp)/2
933         } else if (iedge == 7) {
934             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = +LENY_DOT(iudp)/2;    lenz_dot = 0
935         } else if (iedge == 8) {
936             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = -LENZ_DOT(iudp)/2
937         } else if (iedge == 9) {
938             lenx_dot = 0;    leny_dot = -LENY_DOT(iudp)/2;    lenz_dot = -LENZ_DOT(iudp)/2
939         } else if (iedge == 10) {
940             lenx_dot = 0;    leny_dot = -LENY_DOT(iudp)/2;    lenz_dot = +LENZ_DOT(iudp)/2
941         } else if (iedge == 11) {
942             lenx_dot = 0;    leny_dot = +LENY_DOT(iudp)/2;    lenz_dot = -LENZ_DOT(iudp)/2

```

Listing of train1.c (41)

```

943         } else if (iedge == 12) {
944             lenx_dot = 0;    leny_dot = +LENY_DOT(iudp)/2;    lenz_dot = +LENZ_DOT(iudp)/2
945         } else {
946             status = OCSM_EDGE_NOT_FOUND;
947             goto cleanup;
948         }
949     } else if (entType == OCSM_FACE) {
950         iface = entIndex;
951
952         if (iface == 1) {
953             lenx_dot = -LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = 0;
954         } else if (iface == 2) {
955             lenx_dot = +LENX_DOT(iudp)/2;    leny_dot = 0;    lenz_dot = 0;
956         } else if (iface == 3) {
957             lenx_dot = 0;    leny_dot = -LENY_DOT(iudp)/2;    lenz_dot = 0;
958         } else if (iface == 4) {
959             lenx_dot = 0;    leny_dot = +LENY_DOT(iudp)/2;    lenz_dot = 0;
960         } else if (iface == 5) {
961             lenx_dot = 0;    leny_dot = 0;    lenz_dot = -LENZ_DOT(iudp)/2;
962         } else if (iface == 6) {
963             lenx_dot = 0;    leny_dot = 0;    lenz_dot = +LENZ_DOT(iudp)/2;
964         } else {
965             status = OCSM_FACE_NOT_FOUND;
966             goto cleanup;
967         }
968
969     } else {
970         status = OCSM_ILLEGAL_VALUE;
971         goto cleanup;
972     }

```

Listing of train1.c (42)

```
973     /* return (constant) velocities, accounting for the movement
974        of the CENTER if it set */
975     for (i = 0; i < npnt; i++) {
976         vels[3*i  ] = lenx_dot + xcent_dot;
977         vels[3*i+1] = leny_dot + ycent_dot;
978         vels[3*i+2] = lenz_dot + zcent_dot;
979     }
980
981     /* compute the sensitivities of the area and volume */
982     area_dot = 2 * (LENX_DOT(iudp) * (LENY(iudp) + LENZ(iudp))
983                   + LENY_DOT(iudp) * (LENZ(iudp) + LENX(iudp))
984                   + LENZ_DOT(iudp) * (LENX(iudp) + LENY(iudp)));
985     volume_dot = LENX_DOT(iudp) * LENY(iudp) * LENZ(iudp)
986                + LENY_DOT(iudp) * LENZ(iudp) * LENX(iudp)
987                + LENZ_DOT(iudp) * LENX(iudp) * LENY(iudp);
988 }
989
990 AREA_DOT( iudp) = area_dot;
991 VOLUME_DOT(iudp) = volume_dot;
992
993 status = EGADS_SUCCESS;
994
995 cleanup:
996 #ifdef DEBUG
997     printf("udpSensitivity -> vels=%f %f %f\n", vels[0], vels[1], vels[2]);
998 #endif
999     return status;
1000 }
```

Listing of `train1.c` (43)

```
1001 /*
1002  ****
1003  *
1004  *   freePrivateData - free private data (just an example)
1005  *
1006  ****
1007 */
1008
1009 static int
1010 freePrivateData(void *data)          /* (in)  pointer to private data */
1011 {
1012     int    status = EGADS_SUCCESS;
1013
1014 #ifdef DEBUG
1015     printf("freePrivateData(%s)\n", (char*)(data));
1016 #endif
1017
1018     /* note: this function would not be necessary if we are only calling
1019        EG_free (and then FREEUDPDATA would not be defined above).
1020        It is simply included here to show how one would write
1021        such a function if the allocation was more complicated
1022        than a simple EG_alloc() */
1023
1024     EG_free(data);
1025
1026 //cleanup:
1027     return status;
1028 }
```

- Lines 1009–1028: 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 1018–1022 for more details.

```
1029 /*
1030  ****
1031  *
1032  *   copyPrivateData - copy private data (just an example)
1033  *
1034  ****
1035  */
1036
1037 static int
1038 copyPrivateData(
1039     /*@null@*/void *src,          /* (in)  pointer to source private data */
1040     void **tgt)                 /* (in)  pointer to target private data */
1041 {
1042     int    status = EGADS_SUCCESS;
1043
1044     *tgt = NULL;
1045
1046     if (src == NULL) goto cleanup;
1047
1048     /* note: this function would not be necessary if we are only calling
1049        EG_free (and then COPYUDPDATA would not be defined above).
1050        It is simply included here to show how one would write
1051        such a function if the allocation was more complicated
1052        than a simple EG_alloc() */
1053
1054     *tgt = (void *) malloc(sizeof(char)*(strlen((char*)src)+1));
1055     if (*tgt == NULL) {
1056         status = EGADS_MALLOC;
1057         goto cleanup;
1058     }
1059 }
```

```
1059 #ifdef DEBUG
1060     printf("copyPrivateData(src=%llx, tgt=%llx)\n", (long long)(src), (long long)(*tgt));
1061 #endif
1062
1063     strcpy(*tgt, src);
1064
1065 cleanup:
1066     return status;
1067 }
```


- 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 & 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

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

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

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

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