

Engineering Sketch Pad (ESP) Training

Session 5: Using UDPs, UDFs, and UDCs

John F. Dannenhoffer, III

Syracuse University

Bob Haimes

Massachusetts Institute of Technology

Revised for v1.13





Overview

- Difference between UDPs, UDFs, and UDCs
- Using user-defined primitives (UDPs)
 - list of UDPs shipped with ESP
 - calling a UDP
- Using user-defined functions (UDFs)
 - list of UDFs shipped with ESP
 - calling a UDF
- Using user-defined components (UDCs)
 - list of UDCs shipped with ESP
 - calling a UDC
- Writing a UDC
 - creating the interface
 - example UDC
- Hands-on exercises:
 - reflected cone
 - fuselage



Differences Between UDPs, UDFs, and UDCs (1)

- Users can add their own user-defined primitives (UDPs)
 - create a single solid
 - do not consume any Bodys from the stack
 - are written in C, C++, or FORTRAN and are compiled
 - can be written either top-down or bottom-up or both
 - have access to the entire suite of methods provided by EGADS
 - are coupled into ESP dynamically at run time
- Users can add their own user-defined functions (UDFs)
 - are the same as UDPs, except they consume one or two Bodys from the stack



Differences Between UDPs, UDFs, and UDCs (2)

- Users can add their own user-defined components (UDCs)
 - can be thought of as “macros”
 - consume zero or more Bodys from the stack
 - create zero or more Bodys (onto the stack)
 - are written as .csm-type scripts



UDPs Shipped with ESP (1)

- `bezier` — generate a Bezier Wire, Sheet, or Solid Body from a input file
- `biconvex` — generate a biconvex airfoil
- `box` — generate a (rectangular) Wire, Sheet, or Solid Body centered at the origin (with possibly-rounded corners)
- `csm` — call `OpenCSM` recursively to read a `.csm` file and create a Body
- `ellipse` — generate an ellipse centered at the origin (try to use the `supell` UDP instead)
- `fitcurve` — fit a Bspline curve to a set of points
- `freeform` — generate a freeform Wire, Sheet, or Solid Body from an input file
- `hex` — create a general hexahedron from its corners segments



UDPs Shipped with ESP (2)

- `import` — read a Body out of a `.step` file
- `kulfan` — generate a Kulfan airfoil
- `naca` — generate a NACA 4-series airfoil or camberline
- `naca456` — generate a NACA 4-, 5-, or 6-series airfoil
- `nurbbody` — generate a Body from a series of NURBS
- `parsec` — generate a Parsec airfoil by either specifying Sobieski's parameters or spline parameters
- `pod` — generates a VSP-like pod
- `tt radwaf` — generate a radial waffle, which is useful for creating fuselage structures
- `sample` — used as an example for users who want to create their own UDP



UDPs Shipped with ESP (3)

- `sew` — sew Faces in a step file into a Solid Body
- `stag` — simple turbomachinery airfoil generator
- `supell` — generate a 4-quadrant super-ellipse
- `waffle` — generate a waffle by extruding a 2D group of segments



Calling a UDP

- UDPs are called with a UDPRIM statement

```
UDPRIM    $primetype $argName1 argValue1 \  
          $argName2 argValue2 \  
          $argName3 argValue3 \  
          $argName4 argValue4
```

- \$primetype must start with a letter
- At most 4 name-value pairs can be specified on the UDPRIM statement
- More name-value pairs can be specified in any number of UDPARG statements that precede the UDPRIM statement

```
udparg    $primetype $argName1 argValue1 \  
          $argName2 argValue2 \  
          $argName3 argValue3 \  
          $argName4 argValue4
```

- name-value pairs are processed in order (with possible over-writing)



UDFs Shipped with ESP

- `createBEM` — create a NASTRAN-type built-up-element (BEM) file from Body on Stack
 - `createPoly` — create a TETGEN .poly file between the two BODYS on the top of the Stack
 - `editAttr` — edit the Attributes for the Body on the top of the stack
-
- UDFs are called in exactly same way as UDPs are called



UDCs Shipped with ESP

- `biconvex` — generate a biconvex airfoil
- `boxudc` — similar to the box UDP
- `diamond` — generate a double-diamond airfoil
- `flapz` — cut a (deflected) flap in a Body
- `gen_rot` — general rotation with two fixed points
- `popupz` — pop up a part of the configuration
- `spoilerz` — pop up a spoiler
- `duct` — generate a duct
- `fuselage` — generate a fuselage
- `strut` — generate a strut (between a duct and wing)
- `wing` — generate a wing



Calling a UDC

- UDCs are called with a UDPRIM statement
- \$primetype must start with a slash (/), dollar-slash (\$/), or dollar-dollar-slash (\$\$/)
 - if /, then the UDC file is in the current working directory
 - if \$/, then the UDC file is in the same directory as the .csm file
 - if \$\$\$/, then the UDC file is in ESP_ROOT/udc
- The UDPRIM statement can be preceded by one or more UDPARG statements
- name-value pairs are processed in order (with possible over-writing)



Writing a UDC

- Define the interface
 - input variables (with default values)
 - output variables (with default values)
 - dimensioned variables (which all default to 0)
- Add assertions to ensure valid inputs
- Make sure all “output” variables are assigned values



Example UDC — dumbbell.udc

```
# dumbbell

INTERFACE Lbar      in  0      # length of bar
INTERFACE Dbar      in  0      # diameter of bar
INTERFACE Dball     in  0      # diameter of balls
INTERFACE vol       out 0      # volume

ASSERT      ifpos(Lbar,1,0)    1
ASSERT      ifpos(Dbar,1,0)    1
ASSERT      ifpos(Dball,1,0)   1
SET         Lhalf      "Lbar / 2"

CYLINDER    -Lhalf  0  0  +Lhalf  0  0  Dbar
SPHERE      -Lhalf  0  0    Dball
UNION
SPHERE      +Lhalf  0  0    Dball
UNION

SET         vol        @volume

END
```



Example UDC — jack.csm

```
# jack

UDPARG $/dumbbell Lbar 5.0
UDPARG $/dumbbell Dball 1.0
UDPRIM $/dumbbell Dbar 0.2
SET     foo @@vol
STORE   dumbbell 0 1

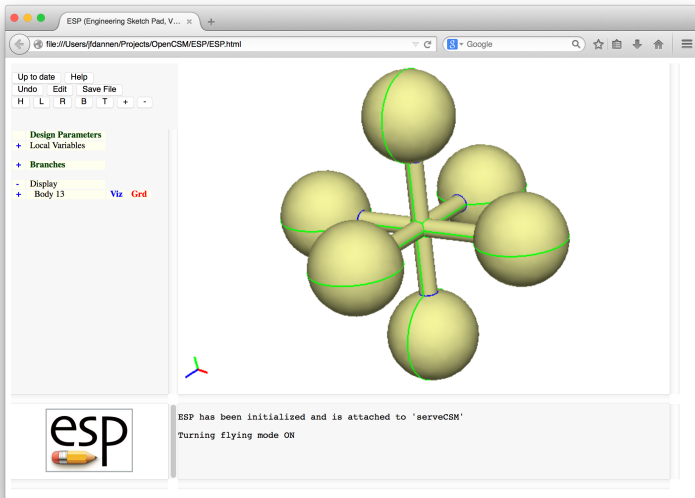
RESTORE dumbbell
ROTATEY 90 0 0
UNION

RESTORE dumbbell
ROTATEZ 90 0 0
UNION

# show that vol was a local variable in .udc
ASSERT ifnan(vol,1,0) 1
END
```



Example UDC — Jack





Example UDC — cutter.udc

```
# cutter
```

```
INTERFACE xx      in  0
```

```
INTERFACE yy      in  0
```

```
INTERFACE zbeg    in  0
```

```
INTERFACE zend    in  0
```

```
ASSERT    ifpos(xx.size-2,1,0)  1
```

```
ASSERT    ifzero(xx.size-yy.size,1,0)  1
```

```
SKBEG      xx[1]      yy[1]      zbeg
```

```
  PATBEG i xx.size-1
```

```
    LINSEG xx[i+1]    yy[i+1]    zbeg
```

```
  PATEND
```

```
    LINSEG      xx[1]      yy[1]      zbeg
```

```
SKEND  1
```

```
EXTRUDE    0  0  zend-zbeg
```

```
END
```



Example UDC — scribeCyl.csm

```
# scribeCyl

DIMENSION xpoints 1 3
DIMENSION ypoints 1 3

SET      xpoints "-1.; 1.; .0;"
SET      ypoints "-.5; -.5; +.5;"

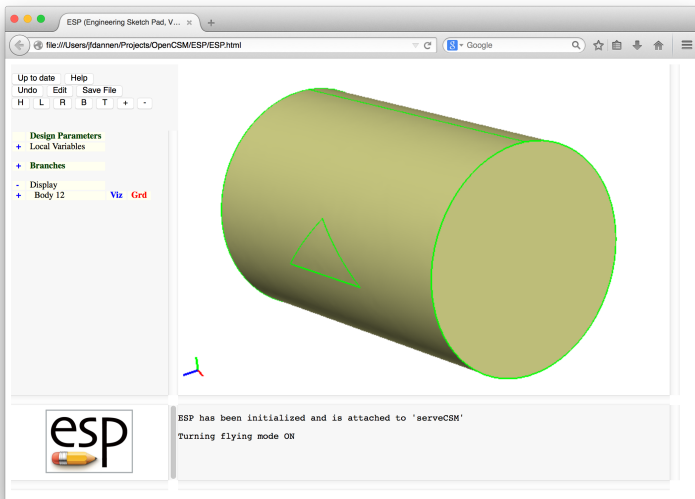
CYLINDER -3 0 0 +3 0 0 2
ROTATEX  90 0 0

UDPARG   $/cutter xx    xpoints
UDPARG   $/cutter yy    ypoints
UDPARG   $/cutter zbeg  0
UDPRIM   $/cutter zend  3
SUBTRACT

END
```



Example UDC — Scribed Cylinder



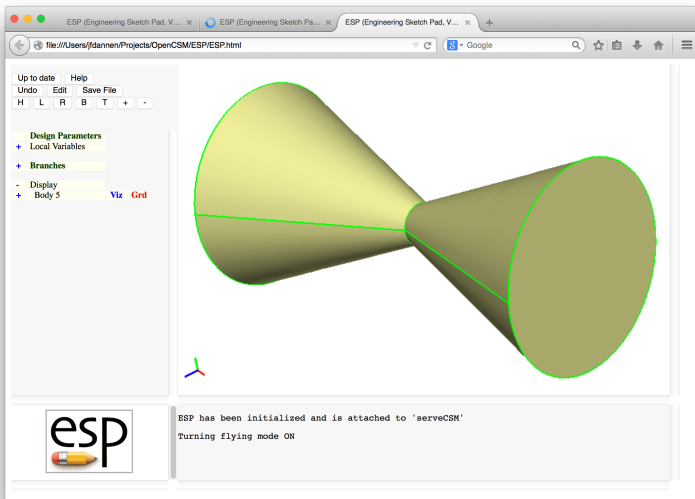


Hands-on Exercises

- Write `mirrorDup.udc` to
 - store a copy of the Body on the top of the stack
 - mirror the Body across a plane whose normal vector and distance from the origin are given
 - union the original and mirrored Bodys
- Apply `mirrorDup.udc` to a cone whose vertex is at the origin
- Write `fuselage.udc` to create a fuselage by blending a series of super-ellipses, where the dimensions of the cross-sections are provided in arrays



Reflected Cone





Muddy Cards

- Any questions?
- Any suggestions?
- Were the examples useful?